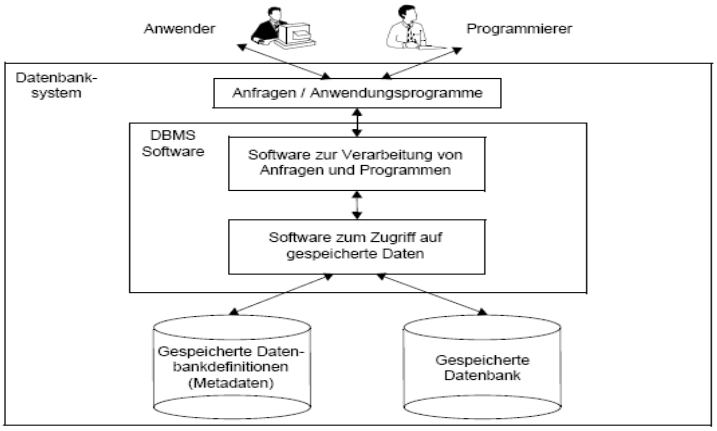
1. Architektur eines Datenbanksystems

1. Konzept des DBS

   Probleme traditioneller Datenverwaltung
   - Redundanz (keine zentrale Kontrolle, anwendungsspezifische Gestaltung der Dateien)
   - Inkonsistenz (Logische und zeitliche Abstimmung der Änderungen in Dateien)
   - Daten-Programm-Abhängigkeit (Änderung des Aufbaus der Datei->Änderung des Programms)
   - Inflexibilität (Daten nicht anwendungsneutral)

   Eine Datenbank ist eine integrierte (vereinheitlichende und anwendungsneutrale Gesamtsicht**), strukturierte (**logisch kohärente Informationseinheiten identifizierbar, redundanzarm/-frei**) Sammlung persistenter Daten, die allen Benutzern eines Anwendungsbereichs als gemeinsame und verlässliche** (Sicherheit der gespeicherten Information) **Basis aktueller Information dient.**
   Ein DBMS ist ein All-Zweck-SW-System, das den Benutzer bei der Definition, Konstruktion und Manipulation von Datenbanken für verschiedene Anwendungen applikationsfrei und effizient unterstützt.
   DBS=DBMS+DB

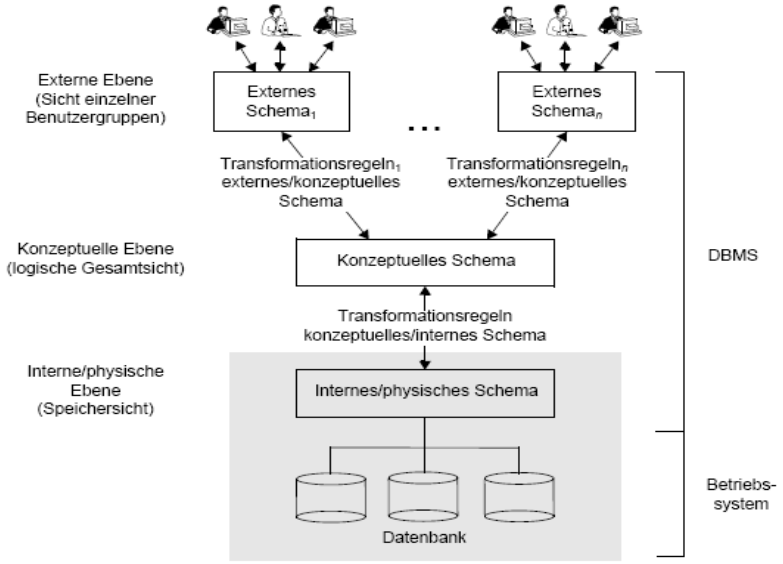


   Jedem DBS liegt ein abstraktes Datenmodell zugrunde, das dem Benutzer eine bestimmte Sicht auf die Daten der Datenbank bietet.
   *Ein **Datenmodell** ist ein mathematischer Formalismus, der aus einer Notation zur Beschreibung der interessierenden Daten und aus einer Menge von Operationen zur Manipulation dieser Daten dient.*
   Ein Datenmodell erlaubt angemessene Beschreibung der Struktur einer Datenbank (Datentypen, Beziehungen, Bedingungen auf Daten). Beispiele: Relationales, objektorientiertes, objektrationales Datenmodell, hierarchisches und Netzwerkmodell.

2. Anforderungen an Datenbanksysteme
   - Datenunabhängigkeit (Anwendungsprogramme unabhängig von Datenrepräsent. und -speicherung)
   - Effizienter Datenzugriff (über Indexstrukturen statt immer Gesamtdurchlauf)
   - Gemeinsame Datenbasis (für alle Anwendungen und Benutzer)
   - Nebenläufiger Datenzugriff (über Transaktionen zur Synchonisation von nebenläufigen Zugriffen)
   - Fehlende oder kontrollierte Redundanz (nur für Aufrechterhaltung logischer Beziehungen oder Performance)
   - Konsistenz der Daten (bei kontrollierter Redundanz)
   - Integrität der Daten (Korrektheit und Vollständigkeit der Daten über Regeln)
   - Datensicherheit (über Views und Zugriffskontrollen- Schutz vor unauthorisierten Zugriffen)
   - Bereitstellung von Backup- und Recoveryverfahren
   - Stellen von Anfragen (über Anfragesprache: effiziente Verarbeitung, lexikalische und Syntaxanalyse, Optimierung, effiziente Ausführung)
   - Bereitstellung verschiedenartiger Benutzerschnittstellen (zB Anfragesprachen für gelegentliche Benutzer, Programmierschnittstellen für Anwendungsprogrammierer, menügesteuerte Schnittstellen..)
   - Flexibilität (Leichte Änderung der Struktur der DB möglich zB neues Feld in Datensätze, als auch einfache Auswertung der Daten nach anderen Gesichtspunkten als bisher)
   - Schnellere Entwicklung von neuen Anwendungen
     Nachteile: Hohe Anfangskosten, Training und Einarbeitung.

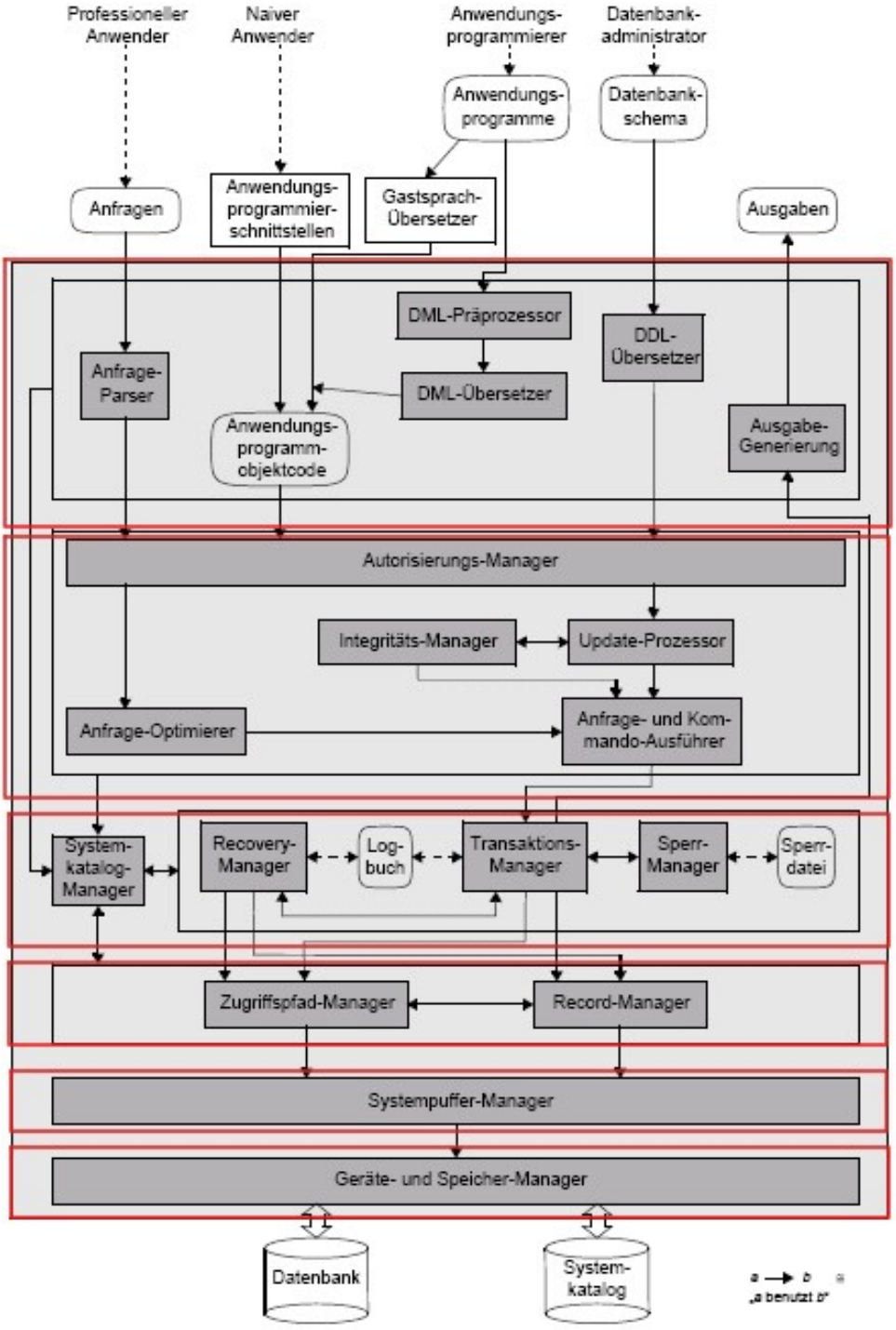
3. Das 3-Ebenen-Modell
   - Phyische/interne Ebene (beschreibt das WIE der Speicherung: Aufbau der Daten, Datenspeicherung, Zugriffspfade)
   - Konzeptuelle Ebene (beschreibt, WAS für Daten gespeichert werden: Logische Gesamtsicht, Abstrahierung von physischen Details. Beschreibt Entitäten, Datentypen, Beziehungen, Integritätsbedingungen)
   - Externe Ebene (umfasst Anzahl von externen Schemata oder Benutzersichten).

   Transformationsregeln definieren Beziehungen zwischen Ebenen:
   Konzeptuell-intern: Beschreibt wie für jedes Objekt des konzeptuellen Schemas die Information aus den physischen Sätzen, Feldern etc erhalten werden kann.
   Extern- konzeptuell: Welche Sicht des konzept. Modells eine Benutzergruppe sieht
   Speicherung der Daten über Schemata der 3 Ebenen + Transformationsregeln (=Meta-Daten) in Systemkatalog.
   Wichtiger Vorteil des 3-Ebenen-Modells: Datenunabhängigkeit (logisch und physisch).
   Vom physischen Modell hängt die Leistungsfähigkeit des gesamten DBS ab!



4. Softwarearchitektur eines DBMS:
   Geräte-und Speichermanager: Verwaltung der Betriebsmittel, verantwortlich für Bereitstellung und Freigabe von Speicherplatz und physische Lokalisierung. Abstrahierung von Zylinderzahl, Spurzahl etc (dies macht das BS oder spezielles System). Angebotene Objekte: Dateien und Blöcke.
   Systempuffer-Manager: Teilt verfügbaren Hauptspeicher in Folge von Seiten oder Rahmen. Abbildung Seite <-> Block und Segment <-> Datei. Lädt Seiten von Sekundärspeicher in Puffer und umgekehrt. Stellt nach oben Segmente mit sichtbare Seitengrenzen im Systempuffer zur Verfügung.
   Zugriffspfad-Manager und Record-Manager: **Zugriffspfad-Manager** verwaltet Anzahl von externen Datenstrukturen zur Abspeicherung/Zugriff von Kollektionen von Datensätzen. **Record-Manager** übernimmt Aufgaben zur internen Darstellung eines logischen Datensatzes. Nach unten: Abbildung interner Datensätze und physischer Zugriffspfade auf Seiten von Segmenten.
   Recovery-Manager, Transaktionsmanager, Sperrmanager: Aufgaben der Synchronisation von parallelen Transaktionen, Nebenläufigkeit, Sperren und Recoverymaßnahmen bei nicht erfolgreich beendeten Transaktionen.
   Autorisierungs-Manager, Integritäts-Manager, Update-Prozessor, Anfrage-Optimierer, Anfrage- und Kommando-Ausführer: Benutzer-Anfragen liegen am oberen Ende der Schicht bereits in interer Zwischenform vor (zB Syntaxbaum). **Autorisierungs-Manager** führt Zugriffskontrolle durch (in Autorisierungstabellen gespeichert), Änderungen werden vom **Update-Prozessor** bearbeitet und Integritätsbedingungen, die bei Definition des Schemas erstellt wurden, werden mit Hilfe des **Integritäts-Managers** überprüft (zB Gehälter>0). Der **Anfrage-Optimierer** optimiert die interne Zwischenform auf Effizienz. **Kommando- und Anfrage-Ausführer** erstellen Ausführungsprogramme bzw. Zugriffspläne, für die Code erzeugt wird.
   Oberste Ebene: Bieten mengenorientierte Schnittstellen an. **DDL-Übersetzer** verarbeitet

Schemadefinitionen, die in einer DDL gespeichert sind und speichert die Beschreibungen der Schemata im Systemkatalog. Lexikalische Analyse durch den **Anfrage-Parser**. **DML-Präprozessor** filtert besonders markierte DML-KOmmandos aus dem Anwendungsprogramm heraus und gibt die dem **DML_Übersetzer,** der sie in Objektcode übersetzt.
Überall vorhanden: **Systemkatalog-Manager**: Steht zu fast allen Komponenten in Verbindung und verwaltet Systemkatalog. Dieser ist auf externem Speicher abgelegt und enthält Meta-Daten über die Struktur der DB. ZB Beschreibung der Daten, Beziehungen untereinander, Konsistenzbedingungen, Zugriffsbefugnisse...



5. Weitere Komponenten eines DBS: zB Abfragesysteme für Laien, Reportgeneratoren für formatierte Berichte, Werkzeuge für Geschäftsgrafiken und CASE-Werkzeuge...

2. Externspeicher- und Systempufferverwaltung

Geräte- und Speichermanager+Systempuffer-Manager bilden das **Speichersystem.**
Erstere verantwortlich für Externspeicherverwaltung. Aufgaben: Speicherung von physischen Datenobjekten auf , Verwaltung freier Bereiche von Sekundärspeichern, Abbildung physischer Blöcke auf externe Speicher, Kontrolle der Datentransports Hauptspeicher <-> Sekundärspeicher. Zweiter verwaltet Systempuffer (=ausgezeichneter Hauptspeicherbereich) und stellt Schnittstelle für das Zugriffssystem bereit- ermöglicht damit Zuordnung von Blöcken zu Dateien, relative Lage der Blöcke zueinander...

2.1 Primär- und Sekundärspeicher

Primärspeicher sind Hauptspeicher und Cache und volatil, Sekundärspeicher persistent. Einheiten zur Speicherung auf Sekundärspeichern: Block (1-8 kB, Vielfaches von 512 Byte).Anordnung auf Spuren (=konzentrische Ringe), diese unterteilt in Sektoren, übereinanderliegende Sektoren=Zylinder. Suchzeit=Pos. des Kopfes auf richtige Spur, Latenzzeit=Wartezeit für Rotation des Kopfes auf richtigen Block, Übertragungszeit=Zeit, für R/W der Daten im Block nach Positionierung des Kopfes. Zugriffszeit=Summe der drei Zeiten.

2.2 Das physische Datenmodell

Physisches Datenmodell beschreibt Implementierungseinzelheiten in Bezug auf Aufbau der Daten, Datenspeicherung und Zugriffspfaden.
Dateien = Zerlegung der Datenbank in disjunkte Teilbereiche.
Vorteile von Dateien:

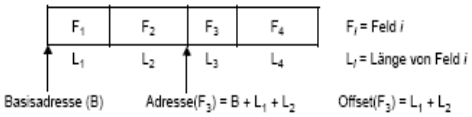
- Nur tatsächlich benötigte Dateien müssen für Zugriff aktiviert werden
- Dateien als dynamische Strukturen ermöglichen Wachsen und Schrumpfen der DB.
- Zuordnung von Dateien auf unterschiedliche Speichermedien möglich für spezielle Anwendungen.
- Kürzere Adresslängen durch Adressierung der Objekte innerhalb einer Datei
- Logisch zusammengehörige Daten (ähnlich/gleich strukturiert) können in einer Datei abgespeichert werden.

Datei logisch= Folge von Datensätzen.
Datei physisch= Folge von Blöcken.
Datensatz=Sammlung von Datenwerten, Beschreibung der Entitäten, ihrer Attribute und Beziehungen.

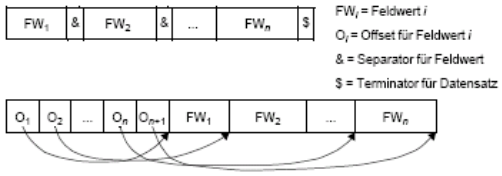
2.3 Datensatzformate

Jeder Datensatz enthält Sammlung von zueinander in Beziehung stehenden Werten. Jeder Wert entspricht einem bestimmten Feld des Datensatzes, in rel. DB ein Feld pro Attribut. Eine Sammlung von Feldnamen und Zuordnung Datentypen <-> Feldnamen legen ein **Datensatzformat** fest.
Alle Datensätze eines gegebenen Datensatzformats haben gleiche Zahl von Feldern.
Ablegen von Datensätzen in Dateien:

- **Datensätze fixer Länge:** Jeder Datensatz hat gleiche Größe in Byte, Adressierung durch Basisadresse+Offset. Nachteil: Speicherplatzverschwendung und schlechte Zugriffszeiten.



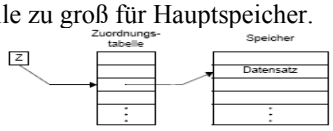
- **Datensätze variabler Länge:** Organisation durch Separatoren und Terminatorsymbol am Ende des Datensatzes oder Längenangabe am Anfang jedes Felds statt Separatoren- Nachteil: Immer Durchlauf nötig. Alternativ: Am Anfang jedes Datensatzes Verzeichnis von Integer-Offsets oder Zeigern. Probleme bei Änderungen:Alle nachfolgenden Felder+deren Offsets müssen verschoben werden.



- **Datensätze sehr großer Länge:** Blobs (binary large objects) passen oft nicht auf eine Seite. Diese können in kleinere Datensätze unterteilt werden, wobei jeder kleinere Datensatz einen Zeiter auf den nächsten kleineren Datensatz enthält=Verkettung. Alternativ: Nur die wichtigsten Infos im Feld eines Datensatzes speichern und Genaueres auf spezielle Datenseiten auslagern.
- A**usrichtung von Feldwerten:** bedeutet, dass absolute Anfangsadresse eines Feldwertes durch die Anzahl der bytes zur Darstellung des zugehörigen Typs teilbar sein muss (dh.

Einschränkung bei allen Typen, die mehr als 1 Byte benötigen zB Integer, Real..). Die absolute Anfangsadresse eines Datensatzes muss durch die größte Zahl an Bytes teilbar sein, die eines seiner Feldwerte zur Darstellung benötigt. Evtl. höhere Effizienz durch Vertauschung der Feldwerte.

- **Zeiger:**
  1. Physische Zeiger: Schnell, aber Verschieben eines DS innerhalb einer Seite nicht möglich (alle Zeiger auf diesen DS müssten gefunden werden)
  2. Seitenbezogene Zeiger: Paar (s,p). p= Indexposition auf Seite s.
  3. Logische Zeiger: Zeiger verweist auf Indexposition einer Zuordnungstabelle, deren Eintrag diePosition des DS im Speicher angibt. Nachteil: Zusätzlicher Zugriff und oft Tabelle zu groß für Hauptspeicher.

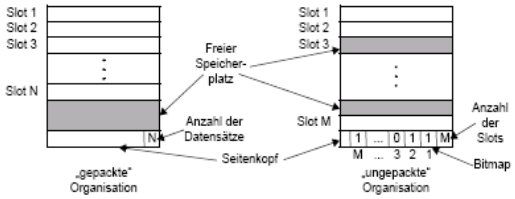


  4. Fixierte Datensätze: Zeiger unbekannter Herkunft zeigen auf Datensätze. Nur unfixierte können innerhalb oder zwischen Seiten verschoben werden, unfixierte nur innerhalb einer Seite bei seitenbezogenen Zeigern bzw im Adressraum der Datei bei logischen Zeigern.
  5. Baumelnde Zeiger: Entstehen, wenn fixierte Datensätze gelöscht werden. Die Zeiger verweisen dann auf falsche oder fehlende DS-> für jeden DS eine Löschmarkierung verwalten, freigewordener Speicherplatz kann dann aber nicht verwendet werden.

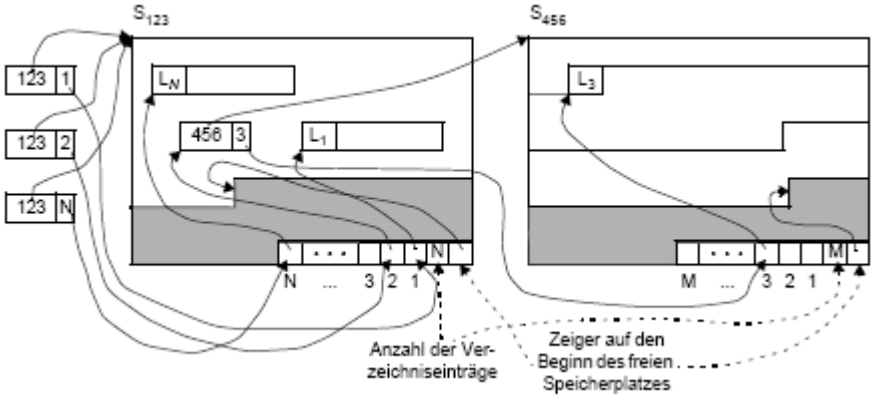
2.4 Seitenformate

Vorstellung: Seite=Sammlung von Slots, jeder Slot enthält einen DS. Ein Datensatzidentifikator (DID) besteht aus Paar (s,n), n verweist auf den Slot.

- Seitenformate für DS fixer Länge:
  1. Gepackte Organisation: Ablage von N DS in die ersten N Slots: Unflexibel
  2. Ungepackte Organisation: Freie Slots werden über Bitmap-Verzeichnis verwaltet. Die Suche des i-ten Datensatzes erfordert Durchlauf durch Verzeichnis. Auf Verzeichnis kann verzichtet werden, wenn es ein spezielles Feld gibt, das einen Zeiger auf den 1.freien Slot enthält. Dieser enthält wiederum einen Zeiger auf den nächsten freien Slot uw.



- Seitenformate für DS variabler Länge: Zur Vermeidung von Fragmentierung (ungenutzer Speicherplatz zwischen DS) müssen DS verschoben werden. Möglich ist bei unfixierten DS die gepackte Organisation plus speziellem Terminatorsymbol oder Längenangabe am Satzanfang. Allgemein aber Indirektion:, am flexibelsten durch das Datensatzidentifikator-Konzept (DID-Konzept), in relationalen DB bekannt als Tupelidentifikator-Konzept (TID): Jedem DS wird ein eindeutiger, stabiler DID zugeordnet, bestehend aus Seitennummer und Index:



Freispeicherverwaltung durch Zeiger auf 1.freien Slot. Ist ein DS zu groß zur Aufnahme in die Seite, wird diese komprimiert->maximaler Speicherplatz. Passt er immer noch nicht hinein->Auslagerung auf Überlaufseite (max 1)->Zugriff mit max. 2 Seitenzugriffen.

- Seitenformate für Datensätze sehr großer Länge: Zwei Strategien: 1. Zerlegung des DS in eine Folge von mittels Zeigern verketteten, kleineren DS und über Anzahl von Seiten verteilt wird (spanned records). 2. Nur die wichtigsten Informationen werden abgespeichert und genaue Repräsentation auf Folge von speziellen Datenseiten.

2.5 Abbildung von Datensätzen in Seiten

Clusterung=Anordnung logische verwandter Datensätze physisch nahe beieinanderliegend.
Clusterung ist wichtiger Faktor für Effizienz, weil dann die zusammengehörige DS entweder schon alle im Systempuffer sind oder zumindest die Suchzeit klein ist, weil der R/W-Kopf kaum bewegt werden muss.
Auch zusammengehörige Dateien können sinnvoll geclustert werden, zB bei Lieferanten- und Warensendungen-Dateien können Zusammengehörige DS verzahnt gespeichert werden.

2.6 Dateien

Operationen auf Dateien: Suche und Änderung.
Operationen zur Lokalisierung und Zugriff auf DS:

- Datensatz auffinden: Op sucht ersten DS, der ein Selektionskriterium erfüllt- laden der Seite in Systempuffer, dort wird der DS lokalisiert und zum aktuellen DS.
- Datensatz lesen: Kopieren des aktuellen DS aus Systempuffer in Variable oder Arbeitsbereich einer Anwendung.
- Nächsten Datensatz finden: Op sucht nächsten DS, der das Selektionskriterium erfüllt, danach weiter wie Datensatz auffinden.
- Datensatz löschen: Löschen des aktuellen DS aus Systempuffer und ggf. aktualisieren der externen Datei.
- Datensatz verändern: Verändern von Feldwerten des aktuellen DS in Seite im Systempuffer und ggf Aktualisieren der externen Datei.
- Datensatz einfügen: Zuerst Seite feststellen, in die eingefügt werden muss, dann Laden dieser Seite in Systempuffer, dort einfügen und ggf Aktualisieren der externen Datei.

Mengen der konzeptuellen Ebene werden auf Dateien gespeichert: Relationale DBMS: Jede Relation<-> einer Datei. In objektorientierten DBMS: Klassen <-> Datei, Objekte<->Datensätze, Attributwerte<->Feldwerte.

2.7. Grundlegende Dateiorganisation

Kostenbetrachtung: b=Anzahl der Dateiseiten, r=Anzahl der DS/Seite (=Blockungsfaktor), d=Zeit zum R/W einer Seite, c= Verarbeitungszeit eines DS, h= Zeit für Ausrechnen einer Hashfunktion.

1. **Haufendateien**
   Einfügen der DS in ungeordneter chronologischer Reihenfolge in heap file. Neue DS ans Ende der Datei.
   Mögliche Realsierungsformen:
   - Doppelt verkettete Listen- DBMS richtet Tabelle mit (Haufendateiname, Adresse der 1.Seite=Kopfseite)-Einträgen ein. Freier Speicherplatz innerhalb einer Seite muss verwaltet werden als auch alle Seiten mit freiem Speicherplatz, zB durch doppelt verkettete Listen für a) freie Seiten und b) volle Seiten.
   - Verzeichnisse von Seiten: DBMS muss sich Ort der 1.Verzeichnisseite des heap file merken.Jeder Eintrag zeigt auf eine Seiten in der Haufendatei. Freispeicherverwaltung mittels Biteintrags oder Zählers pro Eintrag mit Bytezahl des freien Speicherplatzes.

   Durchlauf: Kosten b(d+rc)
   Suche mit Gleichheitsbedingung: 0,5*b(d+rc) falls Bedingung bzgl Schlüsselfeld, sonst gesamte Datei durchsuchen.
   Suche mit Bereichsbedingung: b(d+rc) gesamte Datei muss durchlaufen werden.
   Einfügen eines DS:2d+c letzte Seite laden,DS einfügen und zurückschreiben
2. **Sequentielle Dateien**
   Sortierung auf Basis der Werte eines der Datensatzfelder (=Ordnungs- oder Sortierfeld).Falls dies ein Schlüsselfeld ist, spricht man auch von Sortierschlüssel.
   Durchlauf: b(d+rc)
   Suche mit Gleichheitsbedingung: d log b+c log r – Seite finden + DS auf Seite finden (beides binär). Gibt es mehrere DS für die Bedingung, sind diese auf derselben Seite gespeichert.
   Suche mit Bereichsbedingung: Kosten für Suche nach erstem+ letztem DS für diese Bedingung.
   Einfügen eines DS: 2 x 0,5 b(d+rc)- -richtige Einfügeposition in Datei finden, DS auf der gefundenen Seite platzieren, alle nachfolgenden Seiten laden und zurückschreiben. Sortierreihenfolge nur bei

unfixierten DS aufrechtzuerhalten!
Löschen eines DS: Suchkosten+b(d+rc)- Seite finden, löschen, rückschreiben+nachfolgende DS lesen und rückschreiben zum Heranschieben. Bei Löschmarkierungen nur Suchkosten + c+ d.

3. **Hash-Dateien**
   Idee: Verteilung der DS einer Datei in Buckets in Abhängigkeit vom Wert des Suchschlüssels. Hash-Funktion bildet Wert auf Zahl zwischen 0 bis B-1 ab (Streuspeicherung).
   Verwaltung der Behälter durch ein Behälterverzeichnis=Array von Zeigern, indiziert von 0 bis B-1. Seiten für Behälter i sind durch Zeiger miteinander verbunden. Statische Hash-Dateien haben feste Anzahl von Behältern, falls kein Platz mehr-> Überlaufseite. Dynamische Hash-Dateien haben variable Anzahl von Behältern.
   Durchlauf: 1,25 x b(d+rc) da Seiten nur zu 80% gefüllt.
   Suche mit Gleichheitsbedingung: h+d+0,5*rc- besser als sequentiell!
   Suche mit Bereichsbedingung: 1,25*b(d+rc), da gesamte Datei durchlaufen werden muss.
   Einfügen eines DS: h+2d+rc- Hash ausrechnen+ Seite lesen und rückschreiben+ Alle DS lesen ob DS schon vorhanden.
   Löschen eines DS: h+2d+0,5*rc- im Schnitt nur Hälfte der Seite durchsuchen.
4. **Vergleich der Organisationsformen**
   Haufendatei bietet gute Speichereffizienz und schnelles Einfügen und Löschen, schlecht bei Suchoperationen.Beste Struktur bei kompletten Dateidurchläufen.
   Sequentielle Dateien bieten gute Speichereffizienz und schnelles Suchen, besonders mit Bereichsbedingung, langsam bei Einfügen und Löschen. Nicht real in DBMS vorhanden (B-Baum). Beste Struktur bei Suchen mit Bereichsbedingung.
   Hash-Datei bietet gutes Löschen und Einfügen und besonders Suche mit Gleichheitsbedingung. Kein Suchen mit Bereichsbedingung- beste Struktur bei häufigen Suchen mit Gleichheitsbedingung.

Dateiorga-nisation	Durchlauf	Suche mit Gleichheits-bedingung	Suche mit Bereichs-bedingung	Einfügen	Löschen
Haufen	bd	$0{,}5 \cdot bd$	bd	$2 \cdot d$	Suche + d
Sortiert	bd	$d\log_2 b$	$d\log_2 b$ + # Treffer	Suche + bd	Suche + bd
Gestreut	$1{,}25 \cdot bd$	d	$1{,}25 \cdot bd$	$2 \cdot d$	$2 \cdot d$

2.8 Systemkatalog
Systemkatalog enthält Definition der Struktur der DB.
Beschreibung anhand relationaler Datenbanksysteme:
Enthalten systemweite Infos wie Größe des Systempuffers, Seitengröße, verschiedene Schemata und Transformationsregeln.Schemainfos sind alle Angaben über Definition und Struktur der Daten wie Namen u nd zulässige Wertebereiche, logische Beziehungen, Integritätsregeln... und alle Angaben über Speicherung, Codierung und Auffinden der Daten wie Adreß- und Längenangaben, Feldtypen, Zugriffspfade...
Für jede Relation wird der Relationenname, Dateiname, Dateiorganisation (zB Heap), Attributname und Typ jedes Attributs, Indexname jedes definierten Index und Integritätsbedingungen festgehalten.
Für jeden Index die Struktur, jede Sicht der Sichtname usw.
Zahl der Tupel, Art der Speicherung (clusternd), Zahl der Suchschlüsselwerte.
Alle Meta-Daten werden in speziellen Relationen, den Systemrelationen abgespeichert.Dadurch ist der Systemkatalog eine Art Miniatur-Datenbank.

2.9 Systempufferverwaltung

- **Aufgaben:**
  Ziel: Zahl der Seitenzugriffe reduzieren, indem sie so viele Seiten wie möglich in den Hauptspeicher lädt.
  Systempuffermanager bildet die Schicht, die für den Seitentransfer zwischen Externspeicher und Hauptspeicher verantwortlich ist. Hauptspeicher wird in eine Folge von Rahmen zur Aufnahme jeweils einer Seite eingeteilt. Nach oben hin werden Segmente mit sichtbaren Seitengrenzen als lineare Adressräume im Systempuffer zur Verfügung gestellt.
  Systempuffer-Manager wird nach einer Seite gefragt , lädt diese ggf in den Hauptspeicher und liefert die Hauptspeicheradresse zurück. Danaben auch Protokollierungsaufgaben, um die verwendete Seitenersetzungsstrategie aufrechtzuerhalten.

- **Segment-Konzept mit sichtbaren Seitengrenzen**

  Vorteile des Segment-Konzepts:
  - Es können verschiedene Segmenttypen definiert werden, die unterschiedliche Anforderungen an die Verarbeitung effizient unterstützen.
  - Abbildung Segment<->Datei erlaubt Aufrechterhaltung des Datei-Konzepts.
  - Zusätzliche Einführung der Systempufferschnittstelle erlaubt indirekte Einbringungsstrategien zur Unterstützung von Recovery-Maßnahmen. Ansonsten wäre nur direktes Einbringen von geänderten Datenobjekten möglich (update in place).
  - Segmente können als Einheiten des Sperrens, der Wiederherstellung bei Systemfehlern und der Zugriffskontrolle dienen.

  Beispiele für Segmenttypen:
  - Öffentliche Semente für Speicherung von Datensätzen und Zugriffspfaden incl Nebenläufigkeit und autom. Recovery.
  - Private Segmente zur Speicherung von Log-Infos und Sammlung von statistischen Daten.
  - Temporäre Segmente zur Speicherplatzbereitstellung für kurzzeitige spezielle Anwendungen (zB Sortieren)

  Referenzierung der in den Segmenten gespeicherten Datenobjekten (Datensätze, Einträge):
  a) **Direkte Satzadressierung**: Zugriff auf einen DS mittels seiner relativen Byteadresse innerhalb eines Segments ohne Rücksicht auf Transporteinheiten. Systempuffer-Manager liefert Pufferadresse+Länge des DS zurück. Erfordert die Möglichkeit des Überschreitens von Blockgrenzen durch DS (spanned records= DS wird auf mehrere Seiten verteilt).Nachteile: Erhebliche Ersetzungsprobleme und Beeinträchtigungen von Seitenersetzungsalgorithmen, Sperr- und Log-Komponenten (komplexere Sperrprotokolle und aufwändigere Rückschreibverfahren) und Leistungsverluste.
  b) **Orientierung an den Transporteinheiten des Speichersystems**: Jedes Segment S_k besteht aus geordneter Folge von Seiten P_ki fester Länge L_k. DS können somit unter Beachtung von Seitengrenzen organisiert werden. Spanned records werden nicht unterstützt. Zurückgeliefert wird Pufferadresse der angeforderten Seite.
  Für Feldwerte sehr großer Länge werden die wichtigsten Infos im DS selbst abgespeichert und Genaueres auf eine Folge von Datenseiten ausgelagert- dieser wird dann zusammenhängend in einen Zusatzpuffer geladen, wo er manipuliert werden kann.

- **Abbildung von Segmenten in Dateien**

  Ein Segment wird genau einer Datei zugeordnet und m Segmente können in einer Datei gespeichert werden.
  - **Direkte Seitenadressierung**: Implizit gegebene Zuordnung zwischen Segment und Datei. Meist 1:1-Zuordnung, dh Seite i des Segments S_k wird auf Block i der Datei D_j abgebildet- nur damit dynamisch erweiterbare Segmente. Nachteil: Reservierung des Dateibereichs bei Segmenterzeugung impliziert auch für jede leere Seite einen Block->schlechte Speichereffizienz.

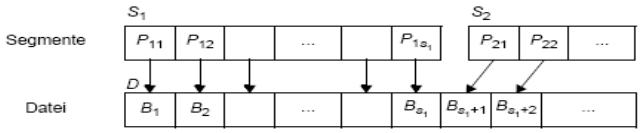


Bild 2.7: Direkte Seitenadressierung

  - **Indirekte Seitenadressierung**: jedem Segment S_k wird eine Seitentabelle T_k zugeordnet mit Eintrag des für jede Seite zugeordneten Blocks. Jeder Datei D_j wird eine Bittabelle M_j zugeordnet für Freispeicherverwaltung, dh. M_j(l) zeigt, ob Block B_jl belegt ist.
  Nachteile: Bei großen Segmenten und Dateien müssen die Seiten- und Bittabellen in Blöcke zerlegt und in speziellem Puffer verwaltet werden. Evtl 2 Zugriffe: Erst Seitentabelle laden um Blockadresse aufzufinden.Wiegt schlechter als die erhöhte Speichereffizienz!

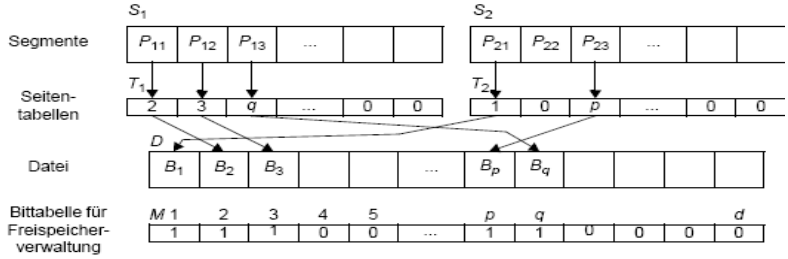
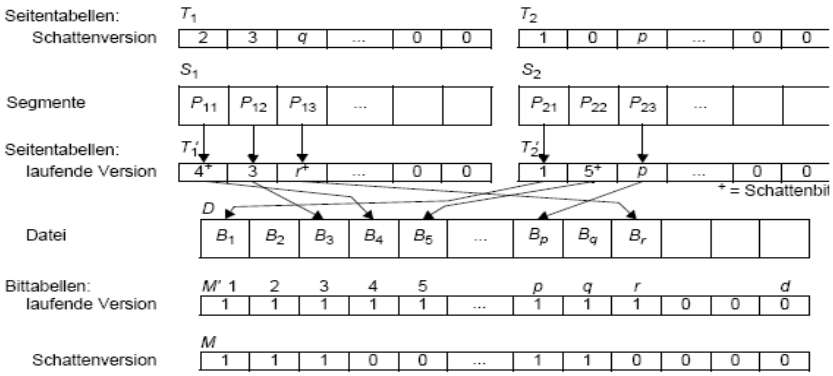


Bild 2.8: Indirekte Seitenadressierung

- **Indirekte Einbringstrategien für Änderungen**
  Um Recovery sicherzustellen, existieren 2 Konzepte für die Erhaltung des alten Zustands, ohne dass aufwändige Log-Informationen geschrieben werden müssen:
  - **Twin Slot-Verfahren**: Modifikation der direkten Seitenadressierung- geringe Kosten für Recovery, doppelter Speicherplatzaufwand.Für eine Seite eines Segments werden jeweils 2 aufeinanderfolgende Blöcke einer Datei reserviert. Der eine hält den aktuellen Zustand fest, auf dem anderen wird geändert. Unterscheidung durch zusätzliche Kennung in der Seite. Bei Seitenanforderung werden beide Blöcke gelesen und der Block mit dem jüngeren Zustand wird im Systempuffer als aktuelle Seite zur Verfügung gestellt.Physisch benachbarte Speicherung zum Lesen beider Seiten in einem Zugriff. Recovery-Konzept mittels Seitensperren ohne Verwaltung von Log-Dateien.
  - **Schattenspeicher-Konzept**: Erweiterung der indirekten Seitenadressierung. Grundidee: Vor Beginn eines neuen, durch 2 Sicherungspunkte gegebenen Sicherungsintervalls den Inhalt aller gerade aktuellen Seiten eines Segments als sog. Schattenseiten duplizieren=> Speicherung einer konsistenten Momentaufnahme des Segments auf Festplatte.Änderungen werden auf Kopien durchgeführt. Geänderte Seiten werden nicht in ihre ursprünglichen, sondern in freie Blöcke zurückgeschrieben.Bei Erzeugung des nächsten Sicherungspunktes werden die Kopien auf Originalplatz rückgeschrieben und die Blöcke freigegeben, deren Seiten während des letzten Sicherungsintervalls einer Änderung unterworfen waren (jüngere Versionen).



  Für s Seiten eines Segments müssen s Blöcke einer Datei als Reserve vorgehalten werden. Keine physische Clusterung möglich.Beträchtliche CPU-Zeit nötig für das Herausschreiben aller geänderten Seiten.
  Sicherungspunkte orientieren sich an Segmenten und nicht an Transaktionen! Daher segmentorientiere Recovery. Für transaktionsorientiere Recovery sind zusätzliche Log-Daten zu sammeln.

- **Verwaltung des Systempuffers**
  Größe zwischen 20 KB und 20 MB.
  Allgemeine Arbeitsweise
  Komponenten ermitteln entsprechende Seitennummern durch Nachsehen im Katalog oder durch Verwendung von Daten über Zugriffspfade, Seite wird angefordert und in Systempuffer geladen, falls sie sich nicht schon dort befindet- ein Rahmen wird gemäß der Ersetzungsstrategie des Systempuffer-Managers ausgewählt, dessen Seite ersetzt wird. Wurde die zu ersetzende Seite geändert, wird sie zuerst rückgeschrieben. Dann wird angeforderte Seite geladen und fixiert, ihre Adresse an Aufrufer übergeben. Durch die Fixierung kann die Seiten nicht aus dem Systempuffer ausgelagert werden und bleibt für die Bearbeitungsdauer im zugewiesenen Rahmen.
  Auffinden einer Seite
  Bei Seitenanforderung hat der Systempuffer-Manager erst festzustellen, ob die Seite bereits im Systempuffer ist, entweder durch direktes Suchen im Pufferrahmen oder indirekt unter Verwendung von Hilfsstrukturen wie Zuordnungstabellen, (un-)sortierten Tabellen, verketteten und Hash-Tabellen. Zuordnungstabelle nur bei kleinen Datenbanken sinnvoll, da (DB-Größe) Seiten zu verwalten sind. Die anderen benötigen nur n Einträge für einen Puffer der Größe n. Sortierte Tabellen benötigen log n Zugriffe, aber hohen Wartungsaufwand beim Einfügen. Verkettete Tabellen geeignet zur Darstellung von bestimmten Seitenreihenfolgen=Seitenreferenzfolgen- Bedeutung für Ersetzungsstrategien (zB LRU). Hash-Verfahren ermöglichen effiziente Suche- ordnet jeder Seitennummer einen Eintrag in der Hastabelle

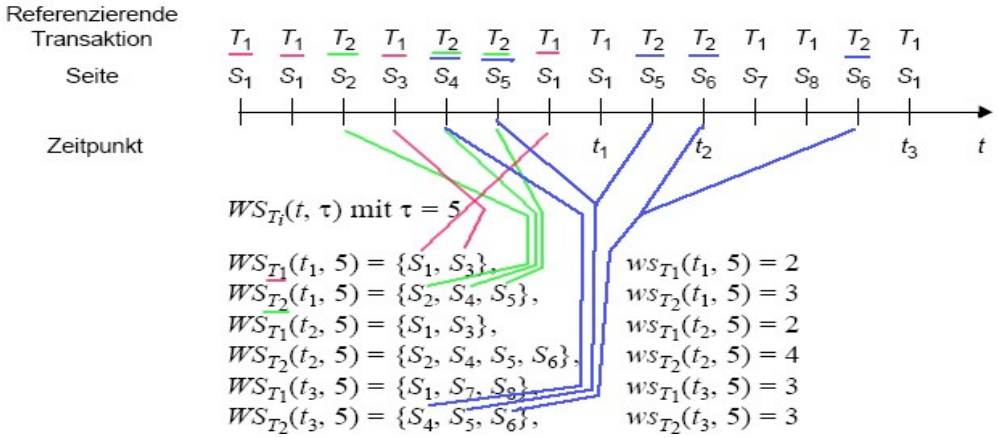
zu, der Seitennummer und Pufferrahmen-Adresse enthält.
Speicherzuteilung im Systempuffer
Speicherzuteilungsstrategie hat die Aufgaben, jeder Transaktion eine Menge von Rahmen zur Aufnahme eines Teils ihrer Seiten zuzuordnen.
Unterschiede zur Optimierung der Speicherzuteilung des BS:

- Zugriff auf eine Seite durch mehrere Transaktionen
- Zugriffe einzelner Transaktionen sind überwiegend sequentiell.
- Zugriffsvorhersage: Für Seiten mit Verwaltungs- und Zugriffspfadsinformationen höher

Unterteilung in lokale, globale und seitentypbezogene Strategien.

- Lokale ordnen jeder Transaktion ohne Einbeziehung des Verhaltens paralleler Transaktionen eine Menge reservierter Rahmen im Systempuffer zu. Weitere Unterteilung:
  - Dynamische Strategien: Transaktion gibt überschüssigen Speicher an andere Transaktionen ab.
  - Statische Strategien: Einmal zugeteilter Speicherplatz fest für Dauer der Transaktion
- Globale teilen den verfügbaren Systempufferrahmen auf alle parallelen Transaktionen unter Einbeziehung ihres gesamten Seitenreferenzverhaltens auf.
- Seitentypbezogene Strategien partitionieren den Systempuffer nach bestimmten Seitentypen wie Datenseiten, Zugriffspfadseiten, Systemseiten..

Bekanntes dynamisches Speicherzuteilungsverfahren: **Working-Set-Strategie**: Der Working-Set WS_T_i(t,tau) einer Transaktion ist die Mengen derjenigen Seiten, die zum Zeitpunkt t von der betrachteten Transaktion innerhalb ihrer letzten tau Seitenzugriffe angesprochen worden sind (tau=Fenstergröße). ws_T_i(t,tau)=|WS_T_i(t,tau)| heißt Working-Set-Größe -je kleiner, desto häufiger benötigte T_i erst kürzlich adressierte Seiten erneut und umso höher war die Lokalität ihres Referenzverhaltens.



Ziel der Working-Set-Strategie: Einer Transaktion T ihren Working-Set verfügbar zu halten. Sie entscheidet, welche Seite zu einem gegebenen Zeitpunkt ersetzbar ist.
Mittels seiten- und transaktionsbezogenen Referenzzählern wird etnschieden, welche Seiten ersetzbar sind: Wird eine Seite S von einer Transaktion T_i referenziert, so wird Trz(T_i) incrementiert und dessen Wert nach srz(T_i,S) kopiert. Ersetzt werden dann die Seiten, für die gilt: trz(T_i)-srz(T_i,S)>=tau. Dies sind die Seiten, die schon lange nicht mehr referenziert wurden.
Beispiel: Zum Zeitpunkt 3 (3.Strich) ist trz(T2)=1 und somit auch srz(T2,S2)=1. Zum Zeitpunkt t3 ist trz(T2)=6 (und srz(T2,S6)=6, da S6 als letztes referenziert wurde). 6-1=5>=tau=5, also ist S2 ersetzbar.

- **Ersetzungsstrategien für Seiten**

Fixierte Seiten sind von der Ersetzungsstrategie ausgenommen.

- FIFO: Älteste Seite wird ersetzt.
- LFU: Ersetzt wird Seite mit der geringsten Referenzhäufigkeit (durch Zähler)
- LRU: Bei jeder Referenz kommt eine Seite an den Anfang der Queue.Schlecht für sequentielle Durchläufe, wenn die Datei mehr Seiten besitzt als der Puffer groß ist->ständige Ersetzungen.
- Clock: Jede Seite hat ein Benutzt-Bit- wird eine Seite eingelagert, werden die Bits überprüft, steht es auf 1 wird es auf 0 gesetzt und mit der nächsten Seite weitergemacht; die erste Seite, deren Benutzt-Bit auf 0 steht, wird ersetzt.
- MRU: Gegenteil von LRU- die am häufigsten referenzierte Seite wird ersetzt.

- Probleme bei der Verwaltung des Systempuffers

Da das DBMS dem BS unterworfen ist, können 3 Fälle entstehen, wenn sich eine benötigte Seite nicht im

Hauptspeicher befindet:

- Page Fault: *Angeforderte* Seite ist im Systempuffer, aber durch das BS ausgelagert- das BS muss wieder einlagern.
- Database Fault: *Angeforderte* Seite nicht im Systempuffer, aber zu ersetzende Seite ist im Hauptspeicher- evtl rückschreiben und dann angeforderte Seite aus der DB auslesen.
- Double Page Fault: *Angeforderte* Seite nicht im Systempuffer , zu ersetzende Seite nicht im Hauptspeicher:BS muss zu ersetzende Seite einlesen, evtl rückschreiben und dann *angeforderte* Seite einlagern.

Bei vielen parallelen Transaktionen kann es zu einer Verknappung an Pufferrahmen kommen->im Extremfall kann **trashing** entstehen (Seitenflattern)- Maßnahmen dagegen: Optimierung der Ersetzungstrategie und des Referenzverhaltens von Programmen sowie Verringerung der Kosten für eine Seitenersetzung. Am effektivsten durch Einschränkung der Parallelität von Transaktionen.

- **Unterschiede der Systempufferverwaltung bei DBMS und BS**
  DBMS kann Seitenreferenzfolgen präziser vorhersagen und muss Seiten fixieren können.
  Des weiteren ermöglicht das **pre-fetching** von Seiten schnelleres Ein-/Ausgabeverhalten.
  Für Recovery muss das DBMS sicherstellen können, dass gewissen Systempufferseiten auf Externspeicher gesichert werden, bevor andere Seiten weggeschrieben werden.

3. Indexstrukturen

Indexe stellen in Dateien abgelegte externe Datenstrukturen dar (B-Baum, Hash-Datei).Sie haben keinen unmittelbaren Einfluß auf den Ort der Abspeicherung, sondern dienen lediglich zum shcnellen Auffinden eines Datensatzes mittels der Adresse. Diese Organisationsformen werden auch als ihre Sekundärorganisationen bezeichnet.
Zuständige Komponente: Der Zugriffspfad-Manager.

3.1 Einführung

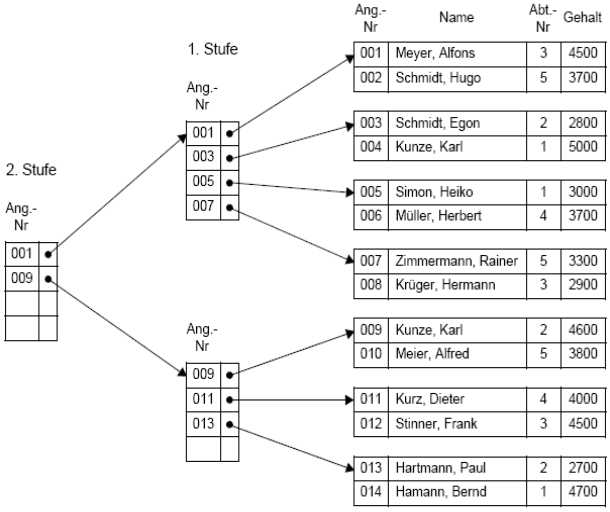
- **Der Begriff der Indexstruktur**
  Allgemein enthält ein Index eine Sammlung von Einträgen, die aus einem Suchschlüsselwert und der Adresse des zugehörigen DS bestehen.
  Nachteile: Ein Index verlangsamt Updates.
  Drei Alternativen für die Struktur eines Indexeintrages k*:
  - k* ist ein Paar (Suchschlüssel k,<Datensatz>)
  - k* ist ein Paar (k,DID)
  - k* ist ein Paar (k,DID-Liste)

  Erste Alternative (d,<Datensatz>) bettet die Zugriffspfadstrukturen in die Speicherungsstrukturen der Datensätze ein, zB duch physische Nachbarschaft oder Zeigerverkettung- spezielle Primärorganisation, benutzbar anstelle einer sortierten Datei oder einer Haufendatei.
  Suchschlüssel: =beschreibt irgendeine Kombination von Attributen, für die man Werte spezifizieren und passende Datensätze erhalten will.
- **Aufgaben von Indexstrukturen**
  Aufgaben einer Indexstruktur: Möglichst schneller Zugriff auf Datensätze anhand eines Suchschlüssels, und Vermeidung eines sequentiellen Durchlaufs.
  Stellen spezielle Operationen zur Verfügung:
  - Sequentieller (sortierter) Zugriff
  - Direkter Zugriff (Punktanfrage)
  - Bereichszugriff
  - Existenztest
- Klassifikationen für Indexstrukturen
  - Dichte und dünne Indexe: Dicht=mind. Ein Indexeintrag/Suchschlüsselwert. Dünn=ein Eintrag für jede Seite.(k,<Datensatz>)-Struktur führt immer zu einem dichten Index, (k,DID) und (k,DID-Liste)-Struktur dicht oder dünn, bei letzterem aber dünner Index nicht sinnvoll. Es kann höchstens einen dünnen Index geben, da dieser eine Sortierung der DS voraussetzt.
  - Primär- und Sekundärindexe: Primärindex=Index, der über Primärschlüssel geordnet ist.
    Einfügen und Löschen: Eintrag auch in Index, bei dünnem Index evtl. auch Änderung der nachfolgenden Indexeinträge erforderlich.
    Siehe Selbsttestaufgabe1.
    Sekundärindexe können nur als dichter Index auftreten, da die Datei nicht physisch nach dem

Sekundärindex sortiert ist. Binäre Suche auf den Indexen ist möglich.
Invertierte Datei: Hat einen dichten Sekundärindex bzgl eines Attributs.
Vollständig invertierte Datei: Datei, die einen dichten Sekundärindex bezüglich jeden Attributs besitzt, das nicht gleich dem Primärschlüssel ist.

- Geclusterte und nicht geclusterte Indexe: Datei so organisiert, dass Ordnung der DS gleich oder beinahe gleich der Ordnung der Einträge ist. (k,<Datensatz>)-Struktur=immer geclustert.Die anderen beiden Strukturen sind nur dann geclustert, wenn DS nach dem Suchschlüsse sortiert sind. Erhaltung geclusterter Indexe extrem teuer, denn anfangs wird auf jeder Seite Speicherplatz für Einfügungen freigehalten und, falls verbraucht, die DS auf Überlaufseiten ausgelagert, nach einiger Zeit wird reorganisiert.
- Indexe mit einfachen und zusammengesetzten Suchschlüsseln: Enthält mehrere Felder. Sortierung 1.Priorität nach dem 1.Feld, 2. Priorität nach dem 2.Feld.
  Unterstützung von Bereichsanfragen:
  - Eindimensionale Indexstrukturen: Lineare Ordnung und Sortierung auf Menge der Suchschlüsseln.
  - Mehrdimensionale Indexstrukturen: keine lineare Ordnung möglich, Organisation der Einträge aufgrund ihrer räumlichen Beziehung. Suche meist langsamer als bei eindimensionalen Indexstrukturen.
- Ein- und mehrstufige Indexe: Binäre Suche auf Index mit b Seiten: 1+log b Seitenzugriffe. Mit Überlaufseiten keine binäre Suche möglich. Idee des mehrstufigen Index: Den noch zu durchsuchenden Index um den Blockungsfaktor r zu reduzieren. Suche in mehrstufigem Index: 1+ log_r b Seitenzugriffe.
  - 1.Stufe=Basisstufe: Aufbau eines dünnen Primärindex
  - 2.Stufe=Index zur 1.Stufe: Ein Eintrag/Seite der 1.Stufe.
  - 3.Stufe=Primärindex für die 2.Stufe, ein Eintrag für jede Seite der 2.Stufe usw.

  Jede Stufe reduziert die Anzahl der Einträge um den Faktor r, wobei r für alle Stufen gleich ist.
  Zahl der Stufen: Für n1 Einträge der 1.Stufe werden k=⌈log_r n1⌉ Stufen benötigt.
  1.Stufe benötigt n2=⌈n1/r⌉ Seiten, 2. Stufe ⌈n2/r⌉ Seiten usw.
  Beispiel: n1=7, r=4:



Um Problemen beim Einfügen zu begegnen->Indexseite nicht vollständig füllen->B-Baum

3.2 Indexstrukturen für alphanumerische Daten (ISAM)

Alphanumerisch=numerisch, String, Bool- einfache Struktur, einfache Operationen. Alphanumerische Indexstrukturen sind idR eindimensional.

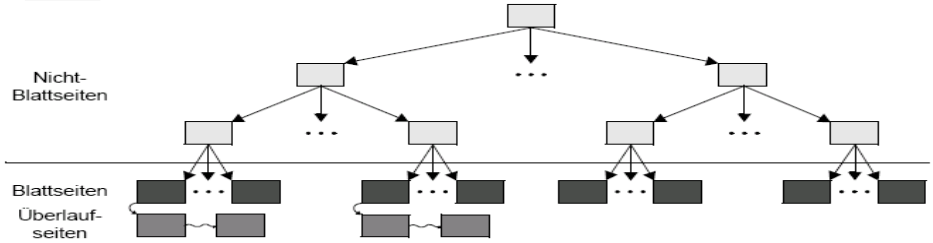
- Index-sequentielle Methode
  Ursprünglich 2-stufiger Index für Festplattenorganisation. Hier mehrstufiger, dünner Index: Datensatzorganisation als sequentielle Datei, darüber ein mehrstufiger dünner Primärindex.
  2 Zugriffsmethoden:
  - 1. Zugriff unter Ausnutzung der sortierten Speicherung
  - Mittels binärer Suche

  Eine index-sequentielle Zugriffsstruktur ist eine statische Blattstruktur und besteht aus Kanten,

Blatt- und Nichtblattseiten. Kante entspricht Zeiger von Seit der Stufe i zu Stufe i+1. Schlüsselwerte eines Knotens sind aufsteigend sortiert.

- Nichtblattseiten beinhalten nur Suchschlüssel=Separatoren zum schnellen Auffinden
- Blattseiten enthalten entweder Datensätze oder DIDs (Indexeinträge auf Datensätze, abgelegt in separater Datei).

Einfügen: Blattseite bereits voll->Allokation einer Überlaufseite, die DS aufnimmt.Daher Seitenauslastung ca 80%, bei Überfüllung Verschlechterung der Suchzeit und Reorganisation nötig.
Wichtiger Vorteil gegenüber B+-Baum: Sperrschritt für Indexseiten kann wegfallen, da diese nie modifiziert werden. Daher Vorteile ggü. B+-Baum bei relativ statischer Datenverteilung und Größe!

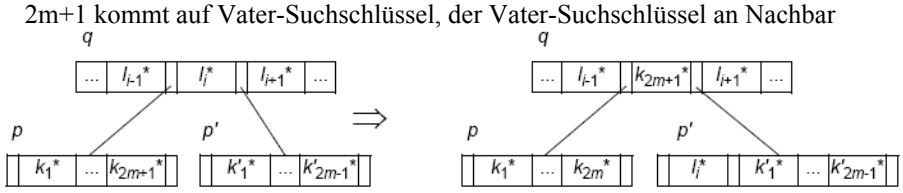


- **Baumbasierte Indexstrukturen: B-, B+, B***

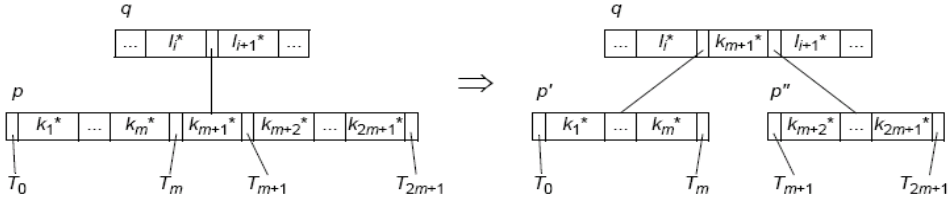
Sind dynamische, mehrstufige Indexstrukturen, effizientes Einfügen und Löschen.

- Vielweg-Suchbäume: sind Verallgemeinerungen des binären Suchbaumes und haben die Struktur T0 l1 T1 l2 T2... wobei T0..Tn Vielweg-Suchbäume mit Suchschlüsselmengen T'0..t'n sind und l1..ln (li<l(i+1)) eine Folge von Suchschlüsseln.
  Es gilt, dass alle Suchschlüsselmengen im Baum links eines Suchschlüssels l kleiner sind als l, rechts davon größer.
- **B-Bäume**: Sind Varianten der Vielweg-Suchbäume- haben Ordnung m. Jeder Knoten enthält Eintrag T0 k1* T1 k2*... wobei die k* Einträge der Form (Suchschlüssel, Datensatz/DID) sind.
  Unterschiede:
  - Jeder Knoten besitzt zwischen m und 2m Suchschlüssel (nur Wurzel 1 bis 2m)
  - Alle Pfade von Wurzel zu einem Blatt sind gleich lang
  - Jeder innere Knoten mit n Suchschlüsseln hat genau n+1 Söhne (=keine leeren Teilbäume)

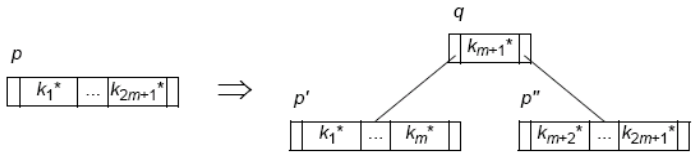
  Einfügen:
  - Wenn Zahl der Suchschlüssel 2m+1 ist=> Overflow
    - Linker oder rechter Nachbar hat <2m Suchschlüssel=>redistribute=der Eintrag an 2m+1 kommt auf Vater-Suchschlüssel, der Vater-Suchschlüssel an Nachbar



    - sonst=> split=Mittlerer Eintrag wird in Vaterknoten eingefügt, falls existent:

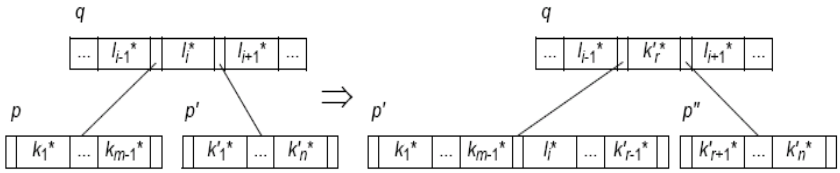


  Evtl weitere Behandlung des Überlaufs bis zur Wurzel.
  Falls kein Vaterknoten existent (wird also in die Wurzel eingefügt)=>neue Wurzel:

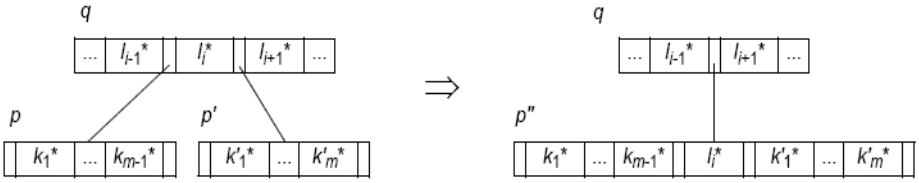


Löschen:

- Falls Unterlauf=>underflow
  - Falls Nachbar existent mit mehr als m Schlüsseln=>balance=Ersetzen des Vaterschlüssels durch mittleren Schlüssels beider benachbarter Knoten, Vaterschlüssel in Sohn einfügen:



  - sonst=>merge=Verschmelzen samt des Vaterknotens:

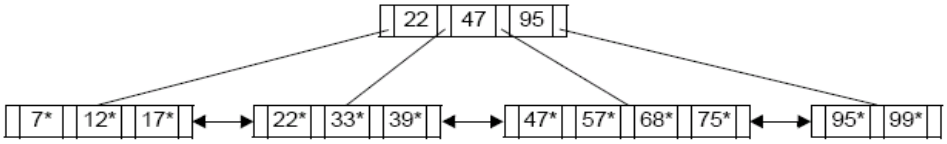


Kosten für alle 3 Ops: O(log_(m+1) n), Speicherplatz O(n). Minimale Speichernutzung von >50%, im Schnitt zu 69%.

- B+-Bäume:

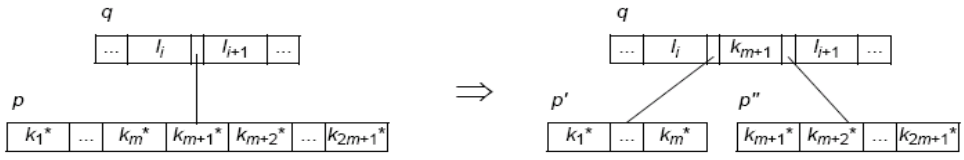
Modifikation zu B-Baum=>

- die inneren Knoten enthalten nur die Suchschlüssel k und die Verweise auf die Teilbäume. Die Suchschlüssel k dienen dann nur als Wegweiser
- Alle Suchschlüsselmengen rechts vom Suchschlüssel sind >= k (und nicht nur >)!
- Benachbarte Blattknoten sind doppelt miteinander verkettet.



Einfügen: Ist Knoten vollständig gefüllt und kein Ausgleich mit Nachbar möglich:

- Mittlerer Knoten wird in Vaterknoten **kopiert (Verschieben ist ja nicht möglich, da in Blättern Einträge der Form k*=(k,Daten/DID) und in den inneren Knoten Einträge der Form k stehen)**:



- Läuft innerer Knoten über=>Verschieben des mittleren Elements in Vaterknoten.

Löschen: Entsteht Unterlauf, wird der kleinste bzw größte Schlüssel des Nachbarn in untergelaufenes Blatt bewegt, Vaterknoten erhält Kopie des verschobenen Schlüssels. Verschmelzen und Ausbalancieren wie bei B-Baum.

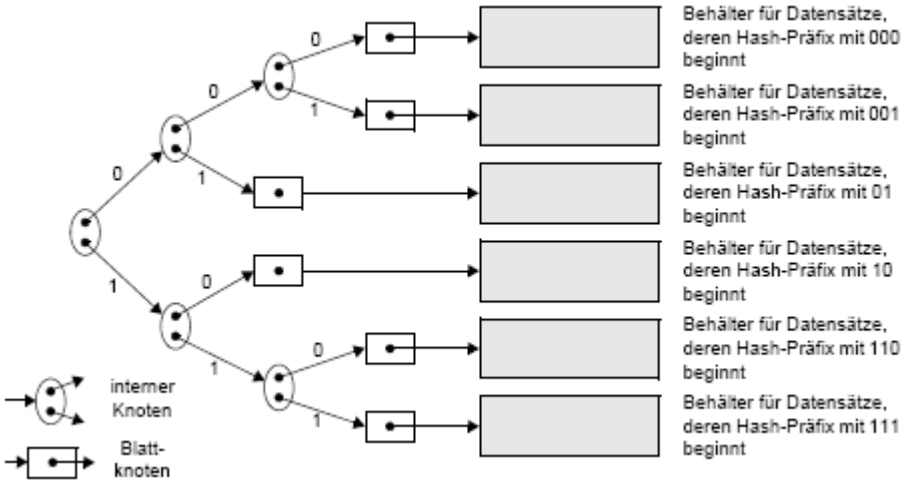
Vorteile des B+-Baumes: Da keine Zeiger auf eigentliche Daten in den inneren Knoten=>mehr Einträge pro Seite möglich als in B-Baum=>evtl geringere Höhe=>verbesserte Suchzeiten.

Aber: Suche führt immer in Blatt, in B-Baum evtl schon in inneren Knoten erfolgreich, und im B+-Baum können Suchschlüsselwerte 2x auftreten

- **Präfix-B+-Bäume**
  Um Höhe des Baumes zu minimieren, und mehr Einträge in die inneren Knoten zu ermöglichen, wird nur ein Präfix des Suchschlüssels verwendet. Jeder Präfix muss größer als der größte Separator in seinem linken und kleiner gleich dem kleinsten Separator in seinem rechten Teilbaum sein.
- **B*-Bäume**
  Statt füllungsgrad von 50-100% in B-Bäumen-> minimaler Füllungsgrad von 70% für jeden Knoten oder unterschiedlich für Blattknoten und innere Knoten.

- **Hash-basierte Indexstrukturen**
  Sehr für das Suchen mit Gleichheitsbedingung geeignet. Es muss kein Pfad durchlaufen werden. Grundlegende Idee: Wert eines Suchschlüssels in einen Bereich von Behälternummern abbilden, um die Seite aufzufinden, die den zugehörigen Indexeintrag (Datensatz/DID) enthält.
  - Statisches Hashing
    Feste Behälternummern, Hashfunktion ist nicht injektiv(für denselben Behälter müssen die Suchschlüsselwerte nicht gleich sein).
    Jedem Behälter sind ein oder wenige Seiten zugeordnet, die als Haufen organisiert sind.
    Verwaltung der Behälter durch Behälterverzeichnis=0 – N-1 indiziertes Array von Zeigern- N sollte deshalb so gewählt sein, dass das Behälterverzeichnis in den Hauptspeicher passt.
    Einfügen: Hashfunktion h anwenden und Behälter suchen, falls Duplikate verboten sind, muss jede Seite des Behälters durchsucht werden, ansonsten auf letzte Seite speichern. Falls dort kein Platz mehr->Überlaufseiten.
    Löschen: Falls der Indexeintrag fixiert ist, wird Löschmarkierung gesetzt, sonst entfernen, ggf kompaktifizieren.
    Durchschnittswerte für N Behälter, n Indexeinträge der Datei, r Indexeinträgen/Seite:
    - Erfolgreiche Suche: $\lceil n/2rN \rceil$
    - Einfügen mit Rückschreiben der Seite: $\lceil n/rN \rceil+1$
    - Löschen/Ändern und Rückschreiben: $\lceil n/2rN \rceil +1$
    - Erfolglose Suche/Überprüfen, dass ein Indexeintrag nicht vorhanden ist: $\lceil n/rN \rceil$
    - Weiterer Zugriff, wenn Behälterverzeichnis nicht in Hauptspeicher.

    Nachteil des statischen Hashing: Wenn Datei schrumpft, wird viel Speicherplatz verschwendet, umgekehrt lange Ketten von Überlaufseiten mit langen Suchzeiten. Reorganisation nötig mit größerem N und Rehashing.
  - Dynamisches Hashing
    Kein festes N, Datei enthält anfangs nur einen Behälter- wird ein 2 Behälter gespalten, sobald er überfüllt ist.
    Das Verzeichnis heißt Digitalbaum.Dieser enthält zwei Arten von Knoten:
    - Interne Knoten, die Separatorfunktion innehaben
    - Zeiger auf einen Behälter

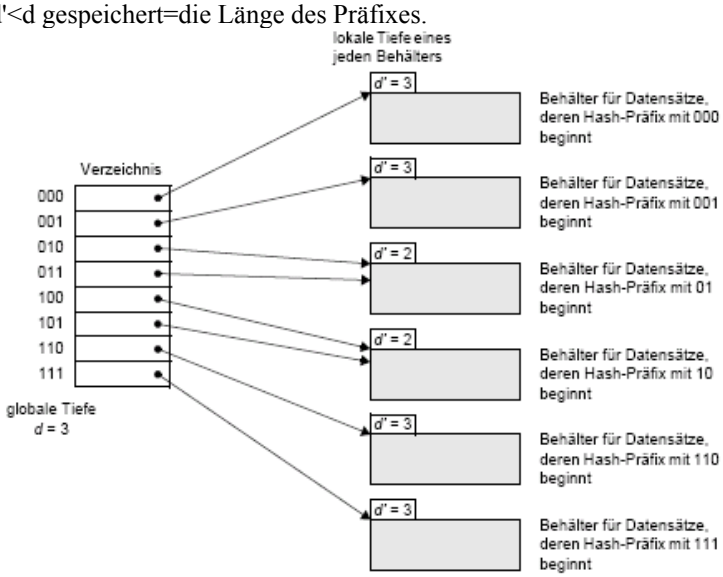


Die Verteilung erfolgt nach dem jeweiligen Bit der Hashfunktion. Falls Behälter überläuft, wird er geteilt und die Indexeinträge gemäß ihres jeweiligen nächsten Bits aufgeteilt. Im Verzeichnis entsteht dadurch ein neuer Knoten.

Es gibt keine Überlaufseiten. Dadurch nur ein Seitenzugriff, falls das Verzeichnis in den Hauptspeicher passt. Vorteil: Sehr speicherplatzsparend.

- Erweiterbares Hashing

Andere Art von Verzeichnisstruktur: Array von 2^d Zeigern auf Behälter mit d=globale Tiefe des Verzeichnisses. Die ersten d Bits der Binärrepräsentation bilden ein Hash-Präfix und legen einen Index eines Verzeichniseintrags fest.
Behälter werden nur auf Anforderung erzeugt (Überlauf), in jedem Behälter wird die lokale Tiefe d'<d gespeichert=die Länge des Präfixes.



Suchen: Die ersten d Bits von h(k) werden als Index betrachtet.
Einfügen: Suchen und Einfügen- falls Überlauf und:

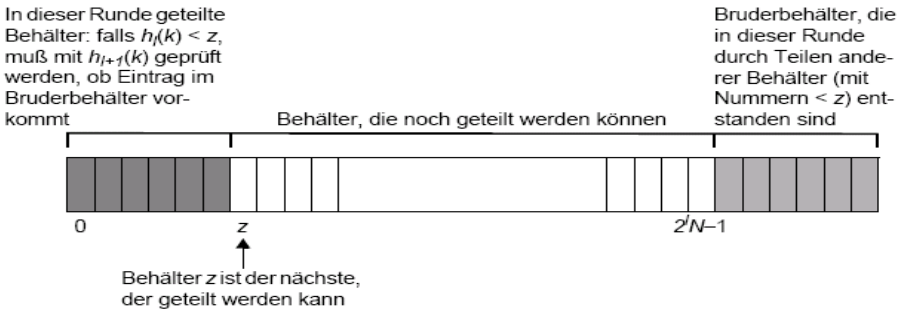
- d=d' => d wird um eins erhöht, und im Verzeichnis wird jeder Eintrag verdoppelt, dh. ihm wird 0 und 1 angefügt (aus 001 entsteht somit 0010 und 0011). Sie zeigen beide auf den bisherigen Behälter (hier also 001). Ein neuer Behälter wird allokiert und der übergelaufene einem Rehashing unterzogen. d' wird mit d aktualisiert. Danach wird eingefügt.
- d>d' => es zeigt bereits mehr als ein Verzeichniseintrag auf den übergelaufenen Behälter->neuer Behälter wird erzeugt, der 2.Zeiger auf diesen gesetzt und jeweils d' um eins erhöht. Dann Rehashing dieser 2 Behälter.

Löschen: Falls 2 Bruderbehälter in einen passen, werden diese verschmolzen. Falls alle d'<d sind, kann die globale Tiefe dekrementiert werden.
Bezogen auf Verzeichnisstruktur ist dynamisches Hashing etwas effizienter (kein doppelten Zeiger). Aber effizientere Speicherung der Verzeichnisstruktur beim erweiterbaren Hashin (Array vs Baum).

- Lineares Hashing

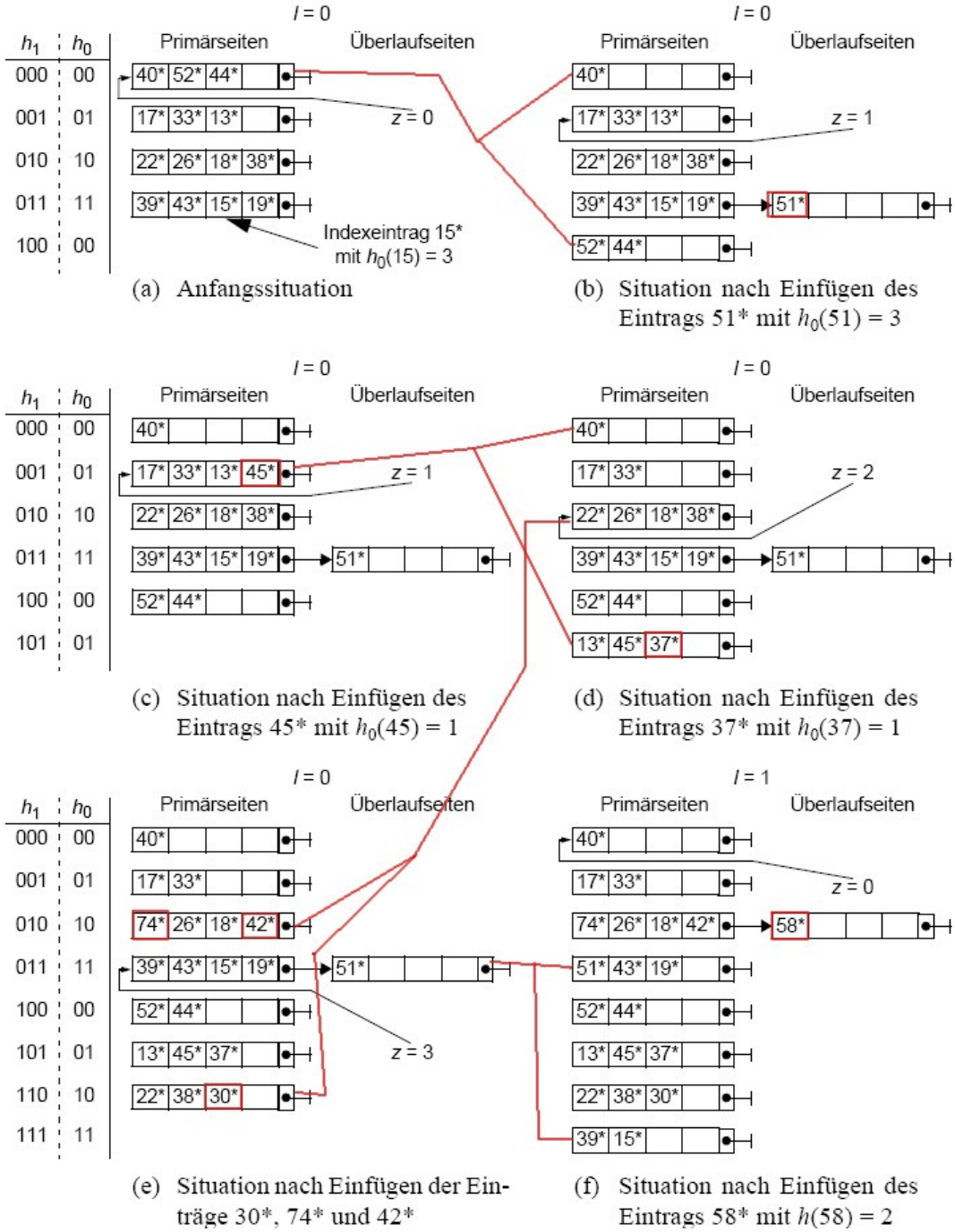
Idee: Dynamisches Wachsen ohne Verwendung einer Verzeichnisstruktur->verschiedene Hash-Funktionen h_0, h_1...mit Eigenschaft: Wertebereich von h_(i+1) =2x Wertebereich von h_i.
Beispiel: h_i(k) = H(k) mod (2^i N) mit N=2er-Potenz=> die letzten d_i Bits werden betrachtet. Es gilt: $d_i = d_0$ +i .
Ist N=4, so ist d_0 =2.

- Rundenzähler l: in Runde l werden nur die Hashfunktionen h_l und h_{l+1} betrachtet. l wird inkrementiert, sobald alle Behälter geteilt wurden.
- Zeiger z zeigt auf den Behälter, der als nächstes geteilt werden kann. Alle Behälter mit $h_l(k) < z$ sind bereits geteilt worden, bei Suche nach einem Eintrag muss bei diesen Behältern zusätzlich geprüft werden mit $h_{l+1}(k)$, ob der Eintrag im Bruderbehälter vorkommt. Falls ein Behälter geteilt werden muss, wird z inkrementiert. Wenn z=N-1 erreicht, wird z auf 0 gesetzt.
  Achtung: **Es wird nicht notwendigerweise der Behälter geteilt, in dem der Überlauf entsteht, sondern der, auf den z zeigt!**



Beispiel für N=4, Behälter z wird geteilt, sobald ein Überlauf stattfindet:

$h_0(40)=h_0(52)=0\,;\, h_1(40)=0\,;\, h_1(52)=4\,.$



Vorteil des linearen Hashings: Bessere Speicherplatzausnutzung durch leichter Beseitigung von größeren Ketten von Überlaufseiten, aber bei schiefer Datenverteilung leidet die Performance durch große Ketten von Überlaufseiten.

- Hash-Funktionen

  Gut entworfene Funktion hat Durchschnittssuchzeit, die konstant und unabhängig von der Anzahl der Suchschlüssel der Datei ist. h muss surjektiv sein, die Werte 0 bis N-1 gleich wahrscheinlich annehmen und jeder Behälter sollte im Schnitt die gleiche Anzahl von zugewiesenen Einträgen enthalten.. Beispiel: Divisionsrestverfahren: $h(k) = k \bmod N$, N mögl. Primzahl.

3.3 Geometrische Indexstrukturen

3.3.1 Einführung

Für spezielle Datentypen reichen konventionelle DB nicht mehr aus, da unzureichende Abfragemöglichkeiten und zu lange Antwortzeiten-> Konzept der erweiterbaren DBS nötig,
Beispiele für Nicht-Standard-Anwendungsbereiche: CAD, VLSI (Geographie, Kartographie).

Geometrische oder räumliche Datenbanksysteme sind speziell auf diese Art von Anwendungen abgestimmt, unterstützen aber auch konventionelle Funkionalitäten.

- **Geometrische Objekte**

  Raumbezogene Objekte enthalten neben thematischen (nicht-geometrischen) Komponenten: Beschreibungen der Gestalt und Lage von im Raum angeordneten, voneinander verschiedenen Entitäten (Städte, Länder..).zB Land: geometrische Beschreibung= innergrenzlichen Fläche, thematische Beschreibung=Bevölkerungszahl.

  Die zu dieser Struktur notwendigen Datentypen werden geometische Datentypen genannt.

  - Punkt- repräsentiert nur Lage
  - Linie- Kurve im Raum
  - Polylinie- Sequenz von Strecken,. Abstraktion von Verbindungen im Raum
  - Region- Abstraktion für 2-dim. Objekt
  - Polygone-disjunkte, geschlossene Polylinie- eine innere und ein/mehrere äußere bilden eine Region.

  Repräsentation geometrischer Objekte=kompliziert und von variabler+ sehr großer Länge.
  Geometrische Algorithmen auf diesen Objekten sind kompliziert und langsam.

- **Geometrische Operationen und Anfragetypen**

  n=Dimension des Raumes, U_i =Universum in Dimension i, $U = U_1 x U_2 x..x U_n$ =n-dimensionales Universum aller raumbezogenen Objekte. $G \subseteq U$. Jedes raumbezogene Objekt $g \in G$ ist gegeben durch

  - das n-dimensionale, minimale,das geometische Objekt umgebende, achsenparallele Rechteck MAR b
  - durch seine exakte geometrische Struktur.

  Für jede Dimension i besteht das MAR b aus Intervall mit linker und rechter Grenze $l_i und\ r_i$.
  =>Durch das n-dimensionale Invervall $[l_1, r_1] x [] x [l_n, r_n]$ wird ein 2n-Tupel repräsentiert: $(l_1, r_{1,..}, l_n, r_n)$

  Folgende Operationen müssen für eine geometrische Indexstruktur IS auf G effizient unterstützt werden:

  - Bereichsanfrage(f,IS(G)): Alle Objekte g der in IS gespeicherten Objekte G sind zu liefern, deren MAR das Anfragefenster f schneidet. Zur effizienten Unterstützung müssen Objekte gemäß ihrer Lage im Raum physisch benachbart auf dem Externspeicher abgelegt werden=> Indexstruktur sollte eine Primärorganisation incl. Physischer Clusterung sein!
  - Enthaltenseinanfrage(f,IS(G)): Liefert alle Objekte g, deren MAR im Anfragefenster f enthalten ist.
  - Nächster Nachbar (p,IS(G)): Liefert das Objekt, das räumlich am nächsten zu Punkt p liegt.
  - Punktsuche(b,IS(G)): Liefert das Objekt g, für dessen MAR g.b gilt: g.b=b
  - Einfügen(g,IS(G)): Ergebnis:IS(G u {g})
  - Löschen(b,IS(G)): Lösche Objekt g mit g.b=b, Ergebnis: IS(G \ {g})

  Punktsuche, Einfügen, Löschen raumunabhängig-können durch B-Baum oder dyn. Hashing effizient unterstützt werden, indem die 2n-Schlüsselkomponenten des n-dim. MAR lexographisch in einem einzigen Schlüssel konkateniert werden.

- **Aufgaben und Eigenschaften geometrischer Indexstrukturen**

  Wichtigste Aufgaben des räumlichen Indizierens: räumliche Selektion, räumlicher Verbund, nächster Nachbar.

  Zwei Stufen des Beantwortens geometrischer Anfragen:

  1. Filterungsschritt: Obermenge der Menge der gewünschten Objekte wird von der DS identifiziert.
  2. Verfeinerungsschritt: Im Hauptspeicher wird jedes von der Indexstruktur gelieferte raumbezogene Objekt einer exakten Überprüfung unterzogen.

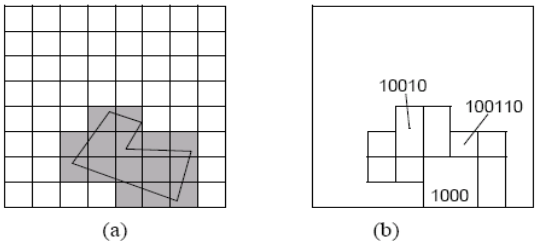
  Die komplexe Geometrie raumbez. Objekte macht eine Approximation derselben durch einfachere Objekte eines Typs fester Länge notwendig. Räumlicher Suchschlüssel ist meist das MAR.
  Strategie einer geometrischen Indexstruktur: Raum+Objekte so zu organisieren, dass bei Suche nur relevante Teile des Raumes betrachtet werden müssen (die Indexstruktur unterstützt ja lediglich den Filterungsschritt). => physische Clusterung nötig.
  Organisation einer geometrischen Indexstuktur: Menge von Behältern, entsprechen Blöcken des Externspeichers. Jedem Behälter wird eine Behälterregion (meist Rechteck) zugeordnet, die einen Teil des Raumes repräsentiert und genau alle im Behälter gespeicherten Objekt enthält. Überlappen der Behälterregionen für Rechteck-DS erlaubt.
  Wachsen einer geometrischen Indexstruktur: Zugehörige Region muss gemäß einer Splitstrategie geteilt werden.
  Schrumpfen: Zwei geeignete Blöcke werden ausgewählt und mit Verschmelzungsstrategie verbunden.

3.3.2 Eindimensionale Einbettungen

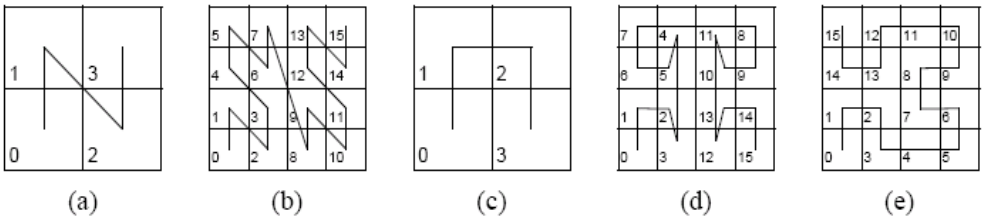
Auf mehrdimensionalen Punkten gibt es keine natürliche Ordnung=> Abbildung auf eindimensionale Indexstrukturen-> Unterteilung des Raums U in reguläres Gitter von Zellen, ein geometrisches Objekt wird durch die Menge der Zellen, die es schneidet, repräsentiert (Gitterapproximation).
Lineare Ordnungen,damit raumlich benachbarte Zellen auch in linearer Ordnung möglichst nah zusammen sind:

- z-Ordnung (a und b): Jeder Quadrant, dem eine Zelle entspricht, wird auf der nächsttieferen Ebene in vier Zellen unterteilt, die in z-Ordnung miteinander verknüpft sind.
  Ermitteln von Zelle 1101 (=13):Rechte obere Zelle hat Bitstring 11 (=3), auf nächsttieferen Ebene ist Zelle 01 die linke obere.
  Andere Sichtweise: Bitverschachtelung (bit interleaving): 1101= 10 x-Koordinate für das 1.+3. Bit, 11y-Koordinate für das 2.+4.Bit.
  Definition der Ordnung durch lexographische Ordnung der Bitstrings.
- Gray-Codierung (c und d)
- Hilberts Kurve (c und e)

Eine Menge von Zellen über einem Gitter mit größtmöglicher Auflösung (=tiefste Ebene der Hierarchie) kann nun in minimale Anzahl von Zellen verschiedener Ebenen zerlegt werden (immer größtmögliche Ebene verwenden s. Bild b),kann also durch eine Menge von Bitstrings (=z-Elemente) dargestellt werden.



Jedem geometrischen Objekt kann seine zugehörige Menge von z-Elementen zugeordnet werden=Menge von geometrischen Schlüsseln. Aufbau des Index durch Vereinigung dieser geometrischen Schlüssel und Ablage in lexographischer Ordnung zB in B+-Baum->Einbettung erhält Nachbarschaftsbeziehungen von Zellen.

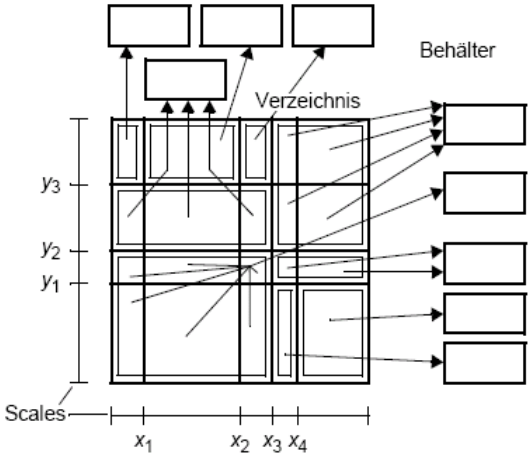


Beispiel: Bereichsanfrage mit Rechteck: Das Rechteck wird in seine z-Elemente zerlegt, für jedes z-Element wird der Teil des B+-Baum durchlaufen, der Einträge besitzt, die mit dem z-Element einen gemeinsamen Präfix haben. Im Verfeinerungsschritt dann exakte Überprüfung der erhaltenen Kandidatenmenge.

3.3.3 Geometrische Indexstrukturen für Punktmengen

- **Das Grid-File**

  Realisiert mehrdimensionales Hash-Verfahren=Verallgemeinerung des eindimensionalen Falls, mehrdimensionaler Raum wird in mehrdimensionale Rechtecke unterteilt, die das Produkt eindimensionaler Intervalle sind=>alle in einem Block gespeicherten Punkte liegen räumlich dicht beieinander, für jede Dimension wird eindimensionales Hash-Verfahren angewandt.
  Beim Grid-File wird der Datenraum, der durch die n Attribute gebildet wird, durch ein orthogonales, unregelmäßiges Gitter unterteilt (nicht Daten, sondern Datenraum wird organisiert).

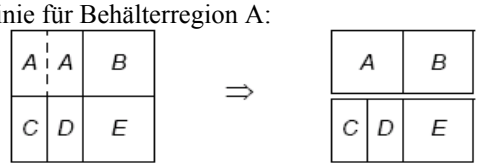


Splitlinien erstrecken sich horizontal oder vertikal durch den gesamten Datenraum. Einteilung des Datenraums durch Scales (ein Scale/Dimension).Ein Verzeichnis (n-dimensionale Matrix) enthält logische Zeiger auf Behälter. Eine Gitterzelle (=kleinster Teilraum des Datenraums) steht in 1:1-Beziehung zu den Elementen der n-dimensionalen Matrix. Alle Punkte innerhalb einer Gitterzellewerden im vom Verzeichnis referenzierten Behälter gespeichert. Es können auch mehrere Gitterzellen ihre Daten im gleichen Behälter ablegen.
Punktsuche: Mittels der Scales Eintrag im Verzeichnis ermitteln, in dessen zugeordneter Gitterzelle der Punkt fällt->Externspeicheradresse des Verzeichniseintrags berechnen, Verzeichnisblock mit dieser Adresse in Hauptspeicher laden->im Verzeichnisblock Externspeicheradresse des zur Gitterzelle gehörigen Behälters ermitteln und Behälter in Hauptspeicher laden=> 2 externe Seitenzugriffe.
Bereichssuche: alle vollständig im Suchbereich enthaltenen Gitterzellen bzw deren Datensätze aus zugehörigem Behälter können übernommen werden, falls Gitterzellen den Suchbereich nur schneiden, müssen die enthaltenen Datensätze einem genauen Test unterzogen werden.
Einfügen und Überlauf: Neuer Behälter wird erzeugt, die Region des übergelaufenen Behälters wird entlang einer Koordinatenachse zerschnitten und beiden Behältern jeweils eine gesplittete Teilregion zugeordnet. Andere dadurch zerschnittene Behälter erhalten zwei Verweise.
Splitstrategie einer Behälterregion:

1. Regel: Längste Seite einer Behälterregion ist zu teilen=>möglichst quadratische Formen erreichen.
2. Regel: Eine Behälterregion ist gemäß einer vorhandenen Einteilung in Gitterzellen zu teilen.=>Verzeichnis nicht unnötig wachsen lassen.
3. Regel: Behälterregion ist in derjenigen Dimension zu teilen, in der die geringste Anzahl von Teilungen erfolgt ist.

Führt keine Regel zu einer eindeutigen Entscheidung=>Behälterregion in beliebiger Dimension teilen.

Teilen einer Verzeichnisblockregion: Gleiche Regeln wie bei Behälterregion + Überprüfen der Einteilung der beiden sich ergebenden Blöcke in Gitterzellen, da sie sich als günstiger erweisen kann.Im Bild Herausnahme der Splitlinie für Behälterregion A:



Löschen: Vorgegebener Füllungsgrad darf nicht unterschritten werden->Vermeiden von ständig wechselnden Teilungs- und Verschmelzungsoperationen. Bei Unterschreiten wird unter Berücksichtigung der Gitterzelleinteilung und der Verschmelzungsstrategie der Bruderbehälter mit der geringsten Füllung ausgewählt.
Verschmelzungsstrategie: Enstehende Behälterregion muss rechteckig sein

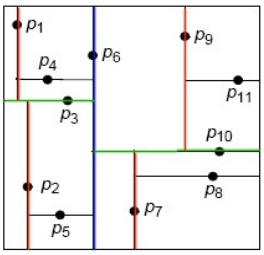
1. Nachbarstrategie erlaubt alle Verschmelzungen, die zu rechteckigen Behälterregionen führen-Deadlocks sind möglich (ungünstige Verschmelzungen, die dann zu keinen weiteren rechteckigen Regionen mehr führen können). Max 4 Verschmelzungspartner.
2. Bruderstrategie: Erlaubt nur Verschmelzen solcher Behälter, die durch Teilung aus einem gemeinsamen Behälter hervorgegangen sein können (=vorherige Teilung wird zurückgenommen). Max 2 Verschmelzungspartner. Garantiert im 2-dimensionalen Fall Deadlock-Freiheit.

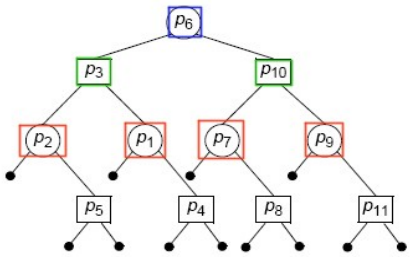
Verschmelzen kann zum Teilen von Verzeichnisregionen führen!

Sinnvoll, Verzeichnis ebenfalls als Grid-File zu organisieren und nicht als externe Matrix=>führt zum hierarchischen oder Mehr-Ebenen-Grid-File->kommt mit nur 2 Ebenen aus, wenn Wurzelverzeichnis im Hauptspeicher gehalten werden kann.

- **Der k-d-B-Baum**

k-d-B-Baum=k-d-Baum + B-Baum.
K-d-Baum= Erweiterung des binären Suchbaumes für mehrdimensionalen Fall->Jede Ebene im Baum unterteilt die Dimension in Teile, also die Wurzel unterteilt Dimension 0 in 2 Datenräume, die beiden Knoten auf Ebene 1 unterteilen die 2 Datenräume auf Dim 1 in 4 Datenräume etc.

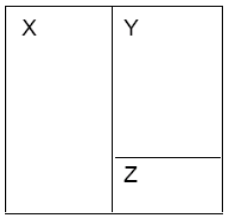




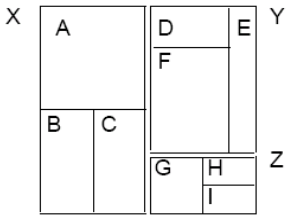
Die Blätter enthalten die eigentlichen Punkte, die inneren Knoten dienen als Separatoren.
Für jeden Knoten im linken Teilbaum eines Knotens gilt: Der Schlüsselwert des durch den Separator im Vaterknoten bestimmten Attributs ist kleiner oder gleich dem Schlüsselwert im Vaterknoten selbst.
Die rekursive Unterteilung des Datenraumes bricht ab, wenn jeder Teilraum nur noch einen einzigen Punkt enthält.
K-d-Baum=typische interne Datenstruktur=> Zuordnung von Teilbäumen des k-d-Baumes auf Externspeicherblöcke durch den **k-d-B-Baum**=Menge von Knoten, wobei jeder Knoten einer Seite entspricht.
Innere Knoten (=Verzeichnis) haben die Form (I,Z), Z=Zeiger auf Nachfolger, I=k-dimensionales Intervall (Region) ($I_o, I_{1,..} I_{k-1}$), enthält alle k-dimensionalen Intervalle des Knotens, auf den Z zeigt.
Blattknoten enthalten Indexeinträge k*, jeder Eintrag bezieht sich auf einen Punkt.
Eigenschaften des k-d-B-Baumes:

- ist ein Vielweg-Suchbaum, keine innere Seite enthält Nullzeiger bzw ist leer
- Für alle Blätter ist die Länge des Pfades von der Wurzel zu einem Blatt gleich
- In jeder inneren Seite sind die k-dimensionalen Intervalle disjunkt, und ihre Vereinigung ist wieder ein k-dimensionales Intervall.
- Ist die Wurzel eine innere Seite, so ist die Vereinigung ihrer k-dimensionalen Intervalle das Kreuzprodukt der Wertebereiche der k Attribute, für die der Baum aufgebaut wurde.
- Ist (I,Z) ein Eintrag in einer inneren Seite, und zeigt Z ebenfalls auf eine innere Seite, so ist I gleich der Vereinigung aller k-dimensionalen Intervalle dieser Seite.
- Ist (I,Z) ein Eintrag in einer inneren Seite, und zeigt Z auf eine Blattseite, so enthält I alle Punkte dieser Seite.

Hier werden aus einer Indexseite 3 Indexseiten durch Partitionierung eines 2-dim.Datenraumes:



Ebene 0 (Wurzel), eine Indexseite



Ebene 1, drei Indexseiten X, Y, Z

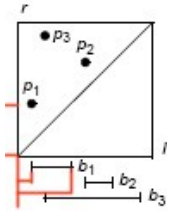
Einfügen: Läuft Blattseite über, wird diese an geeigneter Stelle geteilt (gute Verteilung der Punktobjekte). Split-Fortpflanzung bis zur Wurzel hin möglich.
Forced Split: bei Split von Indexseiten ist evtl Neuaufteilung der Regionen erforderlich (sonst ungünstige Verteilung)-> fortgepflanzter Split nach unten bis zu den Blattknoten wird dadurch ausgelöst.
Suche mit k-dimensionalem Suchintervall S: Rekursiv bis zu den Blättern nach Einträgen (I,Z) suchen für

die gilt: $I \cap S = \emptyset$

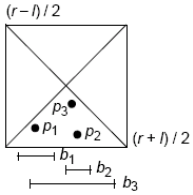
3.3.4 Strategien zur Verwaltung von Rechtecken in externen Datenstrukturen

Speicherung von Rechtecken ggü Punkten schwieriger, weil diese iA Partitionsgrenzen schneiden.

- Transformationsansatz: statt k-dim. Rechtecke 2k-dim. Punkte speichern.
  - Ecktransformation: Bildet m-dim. Rechteck b auf 2m-dimensionalen Punkt p(b) ab. Für ein Intervall b=[l,r] erhält man den Punkt p(b)=(l,r). Da r>=l: alle Bildpunkte liegen oberhalb der Diagonalen=> ungünstige Verteilung, schwere Effizienzprobleme bei externen Datenstrukturen:
    Im Bild ist b_1 die eindimensionale Abbildung des 2-dimensionalen Punktes p_1, wobei sich



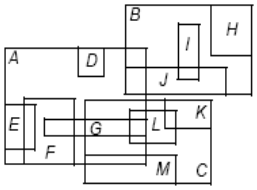
    b_1 bildet aus der x- und y-Komponente von p_1: Die Länge der Strecke ist: $r_1 - l_1$
  - Mittentransformation: Bildet $b = [l_1, r_1] x ... x [l_m, r_m]$ auf den Punkt $p(b) = ((r_1 + l_1)/2, (r_1 - l_1)/2 ..., (r_m + l_m)/2, (r_m - l_m)/2)$ ab -ähnliche Probleme, alle Punkte liegen in einem Kegel.

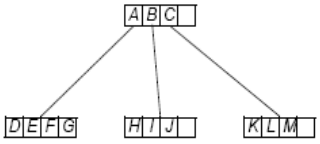


- Überlappen von Behälterregionen: keine Partitionierung zB. R-Baum
- Clipping: Schneidet ein Rechteck eine Grenzlinie der Partition, wird es in mehrere Teile zerlegt und innerhalb eines jeden geschnittenen Teilraums repräsentiert, zB. R+-Baum.

3.3.5 Geometrische Indexstrukturen für Rechteckmengen

- Der R-Baum: Unterschied zum B+-Baum: Teilräume einer Ebene sind nicht notwendigerweise disjunkt.
  Innnere Knoten bestehen aus Einträgen (I,Z), Z Nachfolgerknoten, I=Behälterregion von p=k-dimensionales MAR aller in p abgespeicherten Rechtecken.
  Blattknoten enthalten Indexeinträge k*, jeder Eintrag = Rechteck/abzuspeicherndes Objekt.
  - Zwischen m und 2m Einträgen, ausser Wurzel
  - Eintrag (I,Z) für innere Knoten->I ist das MAR, das alle Rechtecke des durch Z referenzierten Knotens enthält.
  - (I,Z) für Blattknoten->I ist das MAR des durch Z referenzierten geometrischen Objekts.
  - Alle Pfadlängen von Wurzel zu Blatt sind gleich.





Einfügen: Von Behälterregion eines inneren Verzeichnisknotens zu derjenigen im Verzeichnisblock verwalteten Behälterregion der nächst tieferen Ebene, deren Fläche am wenigsten vergrößert werden muss.Läuft der Blattknoten über, wird er möglichst gleichmäßig geteilt (pflanzen sich von unten nach oben fort).
Suche: Suchpfad aufgrund der möglichen Überlappung von Rechtecken nicht eindeutig->evtl mehreren Pfaden folgen.
Bereichssuche: Alle Objekte, für die Z auf einen Blattknoten zeigt, gehören zur Lösung.
Strategien zur Minimierung der Überlapppung: möglichst kleine Flächen bei Teilung entstehen lassen.oder näherungsweise Optimierung der Summe von Umfang, Flächen, Fläche des geometrischen Durchschnitts

der beiden geteilten Regionen.

- Der R+-Baum: Vermeidet das Überlappen von Behälterregionen der gleichen Ebene->Rechteck wird an Partitionsgrenzen in Teile zerlegt und innerhalb jedes geschnittenen Teilraumes repräsentiert.
  Auch hier forced splits möglich.
  Im Bild: G und M sind zu teilen.

4. Externes Sortieren

Wenn die zu sortierenden Daten nicht in den Hauptspeicher passen, wird ein externes Sortierverfahren benötigt. Von Externspeichern kann nur sequentiell gelesen/geschrieben werden, dh. Dass man vorher alle vorhergehenden Datensätze gelesen haben muss, um auf den gewünschten zugreifen zu können.
Wichtig ist Sortieren für:

- Beantwortung von Anfragen in irgendeiner gewünschten Sortierreihenfolge
- Eliminierung von Duplikaten
- Unterstützung der Realisierung relationaler Algebraoperationen wie Join,
- Erzeugung von Partitionen durch Zerlegung einer Datensatzmenge in disjunkte Gruppen

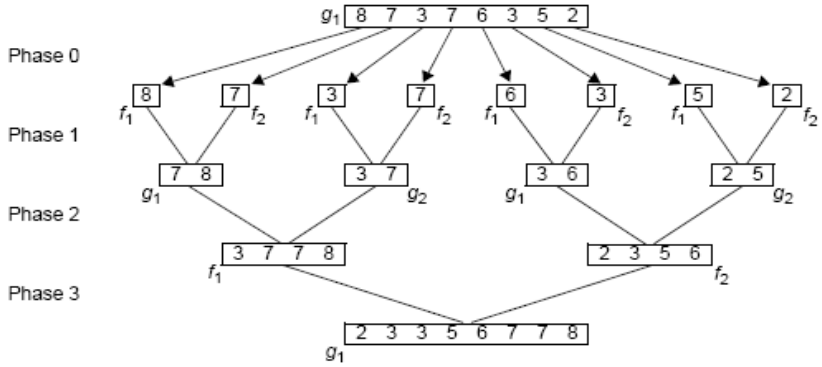
Externe Sortieralgorithmen:

4.1 Direktes und ausgeglichenes 2-Wege-Mergesort

n=Anzahl der DS der Datei.

Direktes 2-Wege-Mergesort-Verfahren

Sortierung geschieht in mehreren Schritten, Misch- oder Verschmelzungsphasen. 4 Dateien werden benötigt! In der Initialisierungsphasen wird die Datei gelesen und Läufe (=aufsteigend geordnete DS) abwechselnd nach $f_1 und\ f_2$ geschrieben. In der Mischphase werden jeweils von den beiden Eingabedaten zwei Läufe parallel gelesen und zu gemeinsamem Lauf verschmolzen:



In jeder Phase werden alle DS jeweils einmal sequentiell gelesen, verarbeitet, geschrieben=>2 Seitenzugriffe/DS.
Anzahl der Phasen: $\lceil \log n \rceil + 1$ =>$2n(\lceil \log n \rceil + 1) = O(\log n)$ Seitenzugriffe.
Bild: $\lceil \log 8 \rceil + 1 = 4$ Phasen und $2x8(\lceil \log 8 \rceil + 1) = 64$ Seitenzugriffe.
Im Hauptspeicher wird nur ein Pufferbereich von 4 Seiten und drei Variable zur Aufnahme der aktuellen DS der beiden Eingabedateien und der gerade gewählten Ausgabedatei benötigt.
Ausgeglichenes 2-Wege-Mergesort:
Beim direkten Mergesort bleibt Hauptspeicher weitgehend ungenutzt=>in der Initialisierungsphase soviele DS in den Hauptspeicher laden , wie dort hineinpassen (m), und mit internem Sortierverfahren (Heapsort/Quicksort) sortieren.=> Anfangsläufe der festen Länge m=>Anzahl der Läufe zu Anfang wird von n auf $k=\lceil n/m \rceil$ reduziert.
Beispiel: Können von 10^6 Datensätzen 10^4 DS intern sortiert werden, so entstehen nur noch 100 Läufe, die in 7 anstelle von 10 Phasen sortiert werden können.
Sei r der Blockungsfaktor (Zahl der DS/Seite), so sinkt die Anzahl der Seitenzugriffe von O(n log n) auf **O((n/r)log(n/m)).**

4.2 Natürliches 2-Wege-Mergesort-Verfahren

Erzeugt Läufe variabler Länge, und nutzt aus, dass meist Teile von Dateien vorsortiert sind und durch eine Methode „Ersetzungsauswahl“ (=Filtern durch einen Heap) Läufe erzeugt werden können, deren Länge die Größe des Hauptspeichers bei weitem übersteigt.
Idee der Ersetzungsauswahl: Anfangslauf erzeugen, der von den m DS im Hauptspeicher den mit dem minimalsten Schlüssel auswählt (zB durch Heapsort) und diesen in die Ausgabedatei schreibt.Er wird dann im Hauptspeicher durch den nächsten DS ersetzt, im Hauptspeicher wird dann wieder der minimalste Schlüssel ausgewählt etc.
Ist kein DS mehr kleiner als der letzte des aktuellen Laufs, wird ein neuer Lauf gestartet.
Dadurch können Läufe erzeugt werden, die größer sind als der Hauptspeicher. Sei m=3:

63 99 47 32 18 89 55 15 96 72 83 25 30

1	2	3	1. Lauf
63	99	47	47
63	99	32	63
18	99	32	99
18	89	32	

1	2	3	2. Lauf
18	89	32	18
55	89	32	32
55	89	15	55
96	89	15	89
96	72	15	96
83	72	15	

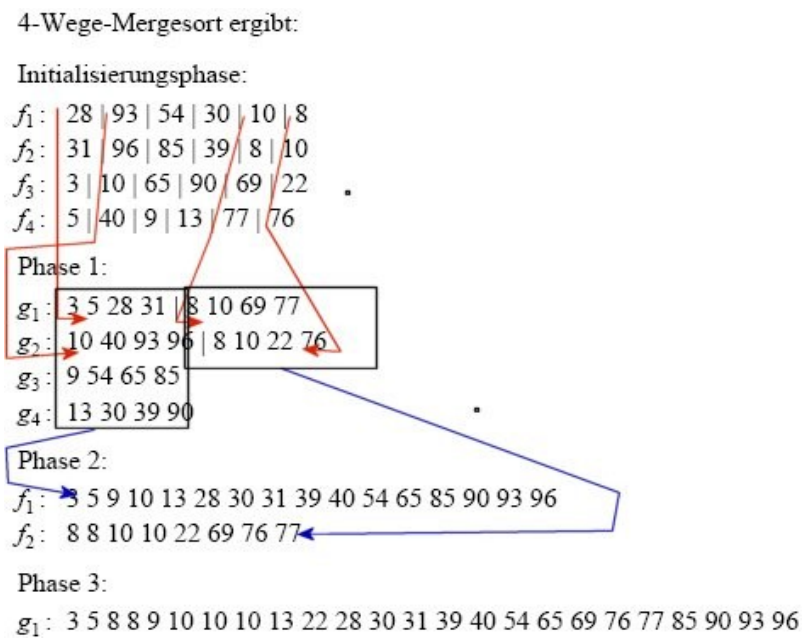
1	2	3	3. Lauf
83	72	15	15
83	72	25	25
83	72	30	30
83	72		72
83			83

Durchschnittliche Länge von Läufen: 2m. Vorsortierungen erhöhen tatsächliche Länge von Läufen erheblich!
Realisierung der Ersetzungsauswahl: Durch Array- von 1 bis i ein Heapbereich h zur Konstruktion des aktuellen Laufs, und i+1 bis m eine Liste von Datensätzen, die auf den nächsten Lauf warten. In jedem Schritt wird nun

1. Der minimalste Schlüssel des Heapbereichs h[1] in die Ausgabedatei geschrieben
2. der nächste DS s mit Schlüssel k eingelesen
   1. ist dieser größergleich k_min=>auf h[1] setzen und in Heap einsinken lassen
   2. ist er aber kleiner als k_min, so muss der Heapbereich um eins reduziert werden- gleichzeitig wird der Schlüssel an h[i] auf h[1] gesetzt und sinkt in heap ein. Auf h[i] wird dann s gesetzt.
3. Nun wird mit 1. fortgefahren, bis der heap leer (=nächster Lauf muss erzeugt werden) oder die Eingabedatei leer ist.

4.3 Ausgeglichenes Mehr-Wege-Mergesort

Verallgemeinerung des ausgeglichenen 2-Wege-Mergesort=>k Ein- und k Ausgabedateien.
Initialisierungsphase: Wiederholt m DS von Eingabeband lesen, intern sortieren und abwechselnd in eine der k Ausgabedateien schreiben=> n/(mk) Läufe der Länge m stehen in den Ausgabedateien.
Mischphase: Ausgabedateien werden zu Eingabedateien und umgekehrt. Die jeweils ersten Läufe werden zu einem Lauf der Länge mk verschmolzen->in die erste Ausgabedatei schreiben. Mit den nächsten Läufen wird genauso verfahren. Dann werden wieder Ein- und Ausgabedatei vertauscht, bis alles vollständig sortiert ist:



In jeder Phase wird Anzahl der Läufe durch k geteilt. Bei l anfänglichen Läufen: $\lceil \log_k l \rceil$ Phasen.
Kosten für Entfernen und Einfügen eines DS in Heap: $O(\log_2 k)$ Zeit.
=> gesamter interner Zeitbedarf/Phase: $O(n \log_2 k)$
Zeitbedarf für alle Phasen: $O(n \log_2 k \log_k l) = O(n \log_2 l)$ => keine Verbesserung ggü binärem Mischen.

4.4 Spezielle Datenbankaspekte beim externen Sortieren

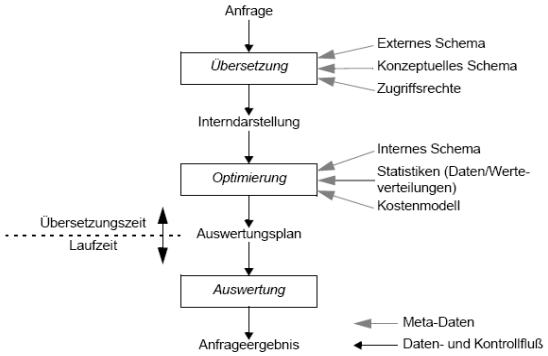
Möglichkeiten der Lage beim ausgeglichenen und natürlichen Mergesort in der Initialisierungsphase im Hauptspeicher:

- Einrichtung eines eigenen Puffers für die Sortierkomponente mit spezieller I/O-Schnittstelle
- Vorhandenen Systempuffer benutzen-Vorteil: Keine Erhöhung der Komplexität.Nachteil: Keine Kontrolle bzgl der physischen Clusterung, hohe I/O-Kosten für eine für Sortieroperationen unnötig flexible Seitenzuordnung (zB Schattenspeicherkonzept), Einschränkung der Effizienz und Optimierungsmöglichkeiten von Sortier-und Mischvorgängen durch Ersetzungsstrategien (LRU) und gemeinsame Benutzung des Systempuffers durch parallele Transaktionen.

Da das Lesen/Schreiben einer Seite genauso viel kostet wie das einer Menge von Seiten b, kann die Anzahl der Externzugriffe um den Faktor b reduziert werden.

5. Anfrageverarbeitung

5.1 Phasen der Anfrageverarbeitung



Drei Phasen:

5.2 Anfrageübersetzung (führt der Anfrage-Parser durch)

Die Anfrage (string) wird in ihre Bestandteile zerlegt, lexikalisch analysiert und auf korrekte Syntax und Semantik geprüft, validiert (Existenz und Zugriffserlaubnis).Dann Überführung in Interndarstellung .

5.2.1 Interndarstellung einer Anfrage

Externe Anfragen (deskriptiv, deklarativ, nicht prozedural) müssen in eine prozedurale Interndarstellung

transformiert werden.zB Relational: Abbildung einer deskriptiven Anfrage auf eine an die Relationenalgebra angelehnte Interdarstellung.
Interndarstellung muss flexibel hinsichtlich Erweiterungen der Anfragesprache und der Durchführbarkeit für nachfolgende Optimierungen sein->Datenstruktur mit effizienten Such- und Zugriffsfunktionen auch auf bestimmte Teile der Darstellung.
Mögliche Interndarstellungen: Relationenkalkül (Tableau-Technik, Zerlegungsbäume, Objektgraphen)- sind nicht prozedural und eignen sich nicht für Erstellung von Auswertungsplänen.
Beispielanfrage:

SELECT *a.Name, a.Beruf*
FROM *Ang a, Abt ab, Proj p, PA pa*
WHERE *a.AbNr = ab.AbNr* AND *ab.AOrt* = "Hagen" AND
*a.ANr = pa.*ANr AND *pa.PNr = p.PNr* AND *p.POrt* = "Hagen"

Übersetzung Ü1 in Relationenalgebra:

$$\pi_{Ang.Name,\ Ang.Beruf}\Big(\sigma_{Ang.AbNr = Abt.AbNr \wedge Abt.AOrt = \text{"Hagen"} \wedge Ang.ANr = PA.ANr \wedge PA.PNr = Proj.PNr \wedge Proj.POrt = \text{"Hagen"}}(Ang \times Abt \times Proj \times PA)\Big)$$

Eine SQL-Anfrage wird in eine Menge von kleineren Einheiten (=Anfrageblöcken) zerlegt und jeder Block für sich übersetzt. Ein Block= eine SQL-Anfrage ohne Verschachtelung, enthält genau eine SELECT-, eine FROM-, höchstens eine WHERE-, eine GROUP BY- und eine HAVING-Klausel.
Die allgemeine Form einer SQL-Anfrage lautet:

SELECT	$r_{i_1}.A_1, \ldots, r_{i_t}.A_t$		SELECT	$R_{i_1}.A_1, \ldots, R_{i_t}.A_t$
FROM	$R_1\ r_1, \ldots, R_k\ r_k$	bzw.	FROM	$R_1, \ldots, R_k$
WHERE	F		WHERE	F

Eine Übersetzung Ü2 dieser allgemeinen Form in Relationenalgebra:

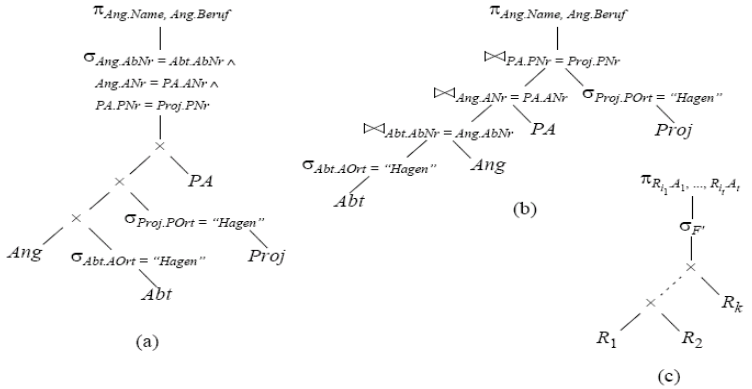
$$\pi_{R_{i_1}.A_1,\ \ldots,\ R_{i_t}.A_t}\left(\sigma_{F'}\left(R_1 \times \ldots \times R_k\right)\right)$$

=kartesisches Produkt aller Relationen in FROM bilden, gemäß der WHERE-Klausel selektieren und auf die Attribute von SELECT projizieren.=> nicht effizient, da erst das kartesische Produkt aller Relationen gebildet wird und erst danach Selektion und Projektion.
Besser: Ü3 und Ü4:

$$\pi_{Ang.Name,\ Ang.Beruf}\Big(\sigma_{Ang.AbNr = Abt.AbNr \wedge Ang.ANr = PA.ANr \wedge PA.PNr = Proj.PNr}\big(Ang \times \sigma_{Abt.AOrt = \text{"Hagen"}}(Abt) \times \sigma_{Proj.POrt = \text{"Hagen"}}(Proj) \times PA\big)\Big)$$

$$\pi_{Ang.Name,\ Ang.Beruf}\big(\sigma_{Abt.AOrt = \text{"Hagen"}}(Abt) \bowtie_{Abt.AbNr = Ang.AbNr} Ang \bowtie_{Ang.ANr = PA.ANr} PA \bowtie_{PA.PNr = Proj.PNr} \sigma_{Proj.POrt = \text{"Hagen"}}(Proj)\big)$$

=> Bzgl der Relation Abt werden nur Abteilungen in Hagen und bzgl der Relation Proj werden nur Projekte in Hagen berücksichtigt.
Ü1-4 stellen Stringtransformationen dar->für die Optimierungsphase inflexibel und ineffizient=> normalerweise interne Darstellung eines relationalen Algebraausdrucks als Operatorbaum- innere Knoten =Operatoren, Blätter=Basisobjekte.

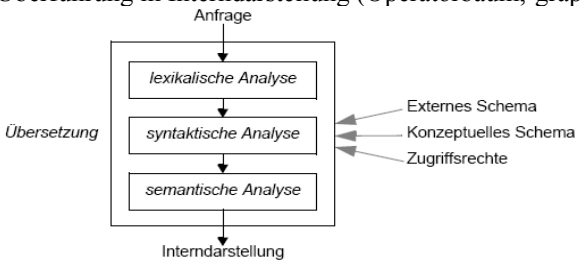


A=Ü3, b=Ü4, c=Ü2

5.2.2 Vorgehensweise bei der Übesetzung

3 Phasen:

1. Lexikalische Analyse=Erkennen der Tokens (=Schlüsselwörter zB SELECT...)
2. Syntaktische Analyse=Korrektheit hinsichtlich der Sprachgrammatik
3. Semantische Analyse=Prüfen der Existenz und Gültigkeit der Objektnamen und Operationen, Überführung in interne Namen, Zugriffsberechtigung, Typüberprüfung der Argumente der Operatoren und Prädikate- danach Überführung in Interndarstellung (Operatorbaum,-graph)

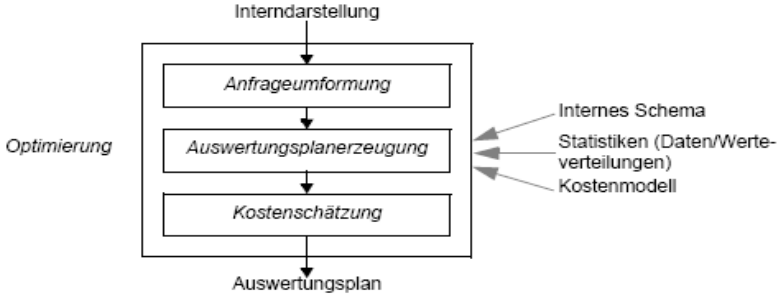


5.3 Anfrageoptimierung

Bestimmt Ausführungs/Zugriffsplan für das WIE (zB durch Indexe) der Ausführung der Anfrage. Zuständig: Anfrage-Optimierer.

3 Teilphasen:

- **Anfrageumformung**- Standardisierung der Interndarstellung in eine normalisierte Form->effizientere Darstellung der Anfrage+Vereinfachung der Anfrageinterndarstellung->Eliminierung von Redundanzen.
- **Auswertungsplanerzeugung**- Abbildung auf alternative Folgen von Elementaroperationen, von denen die effizientesten und von Kosten her bekannten herausgefiltert werden.
- **Kostenabschätzung**- der billigste Auswertungsplan wird bestimmt.



5.3.1 Anfrageumformung

Ziele: Anfragestandardisierung, Anfragevereinfachung durch Äquivalenzumformungen, Anfrageverbesserung durch Restrukturierung (günstige Operatorreihenfolge).

- **Anfragestandardisierung und -umformung**
  Ziel: Überführen in normalisierte Form.
  Für die Verarbeitung sind bestimmte Standardsituationen zu erkennen. Zwei Ebenen sind zu unterscheiden:
  - **Standardformen für den Qualifikationsteil**
    Diese sind konjunktive und disjunktive Normalform- beschreiben den Qualifikationsteil als Konjunktion von Disjunktionsprädikaten bzw Disjunktion von Konjunktionprädikaten. Anfrage in disjunktiver Normalform wird als Vereinigung unabhängiger konjunktiver Unteranfragen verarbeitet- parallele Verarbeitung möglich, kann aber zu duplizierten Join- und Selektionsprädikaten führen.

Kommutativregeln:	$a \wedge b \Leftrightarrow b \wedge a$	$a \vee b \Leftrightarrow b \vee a$
Assoziativregeln:	$(a \wedge b) \wedge c \Leftrightarrow a \wedge (b \wedge c)$	$(a \vee b) \vee c \Leftrightarrow a \vee (b \vee c)$
Distributivregeln:	$a \wedge (b \vee c) \Leftrightarrow (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$
De Morgan'sche Regeln:	$\neg(a \wedge b) \Leftrightarrow \neg a \vee \neg b$	$\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$
Doppelnegationsregel:	$\neg(\neg a) \Leftrightarrow a$	

- **Standardisierung auf der Anfrageebene**
  Bedeutet häufig das Erzeugen einer Prenexform (keine quantifizierten Unteranfragen befinden sich in deren Qualifikationsteil)->alle quantifizierten Ausdrücke werden in Joins umgewandelt:

$$a \wedge \exists r \in rel : p(r) \Leftrightarrow \exists r \in rel : a \wedge p(r) \qquad a \vee \forall r \in rel : p(r) \Leftrightarrow \forall r \in rel : a \vee p(r)$$

$$a \wedge \forall r \in rel : p(r) \Leftrightarrow \forall r \in rel : a \wedge p(r) \quad \text{falls } rel \neq \emptyset$$
$$\Leftrightarrow a \quad \text{falls } rel = \emptyset$$

$$a \vee \exists r \in rel : p(r) \Leftrightarrow \exists r \in rel : a \vee p(r) \quad \text{falls } rel \neq \emptyset$$
$$\Leftrightarrow a \quad \text{falls } rel = \emptyset$$

$$\exists r_1 \in rel_1 \exists r_2 \in rel_2 : p(r_1, r_2) \Leftrightarrow \exists r_2 \in rel_2 \exists r_1 \in rel_1 : p(r_1, r_2)$$

$$\forall r_1 \in rel_1 \forall r_2 \in rel_2 : p(r_1, r_2) \Leftrightarrow \forall r_2 \in rel_2 \forall r_1 \in rel_1 : p(r_1, r_2)$$

$$\exists r \in rel : p(r) \vee q(r) \Leftrightarrow (\exists r \in rel : p(r)) \vee (\exists r \in rel : q(r))$$

$$\forall r \in rel : p(r) \wedge q(r) \Leftrightarrow (\forall r \in rel : p(r)) \wedge (\forall r \in rel : q(r))$$

$$\neg(\forall r \in rel : p(r)) \Leftrightarrow \exists r \in rel : \neg p(r) \qquad \neg(\exists r \in rel : p(r)) \Leftrightarrow \forall r \in rel : \neg p(r)$$

Bild 5.6: Äquivalenzumformungsregeln für quantifizierte Ausdrücke

- **Anfragevereinfachung**
  Aufgabe: Eliminieren von Redundanzen und Inkonistenzen durch Idempotenzregeln:

$$a \wedge a \Leftrightarrow a \qquad a \vee a \Leftrightarrow a \qquad a \wedge true \Leftrightarrow a \qquad a \vee false \Leftrightarrow a$$

$$a \wedge false \Leftrightarrow false \qquad a \vee true \Leftrightarrow true \qquad a \wedge \neg a \Leftrightarrow false \qquad a \vee \neg a \Leftrightarrow true$$

$$a \wedge (a \vee b) \Leftrightarrow a \qquad a \vee (a \wedge b) \Leftrightarrow a$$

$$\sigma_p(\emptyset) \Leftrightarrow \emptyset \qquad \exists r \in \emptyset : p(r) \Leftrightarrow false \qquad \forall r \in \emptyset : p(r) \Leftrightarrow true$$

Bild 5.8: Äquivalenzumformungsregeln für Ausdrücke mit leeren Relationen

- **Anfrageverbesserung durch Anfragerestrukturierung**
  Die standardisierten, vereinfachte, aber noch nicht eigentlich optimierte Interndarstellung einer Anfrage unter Aufrechterhaltung der Anfragesemantik wird in eine äquivalente, optimierte Interndarstellung transformiert.
  Im relationalen Modell: algebraische Transformationen auf Operatorbäumen.
  Methoden der Anfragerestrukturierung stützen sich auf bekannte Heuristiken (=Daumenregel zur Problemlösung)- nicht immer die optimale Lösung, evtl auch negativ.
  Die bekannteste Heuristik: Selektion und Projektion wird vor Join oder anderen binären Ops ausgeführt (da Selektion und Projektion die Größe der einzelnen Relation idR reduzieren und niemals erhöhen).
  Grundlegende heuristische Regeln (R,S,T=Relationen, F=Prädikat),:

1. Linksassoziativität komplexer Mengen-,Join- und Produktoperationen
   Sei $\Diamond \in \{\cup, \cap, \times, \bowtie, \bowtie_F\}$. Dann gilt $R \Diamond S \Diamond T \Leftrightarrow (R \Diamond S) \Diamond T$
2. Kommutativität von Vereinigung und Durchschnitt
   Sei $\Diamond \in \{\cup, \cap\}$. Dann gilt $R \Diamond S \Leftrightarrow S \Diamond R$
3. Kommutativität von Join und kartesischem Produkt.
   Sei $\Diamond \in \{\times, \bowtie, \bowtie_F\}$. Dann gilt $R \Diamond S \Leftrightarrow S \Diamond R$
4. Assoziativität von Vereinigung, Durchschnitt, Join und kartesischem Produkt
   $(R \Diamond S) \Diamond T \Leftrightarrow R \Diamond (S \Diamond T) (R \bowtie_{F_1} S) \bowtie_{F_2} T \Leftrightarrow R \bowtie_{F_1} (S \bowtie_{F_2} T)$
5. Folge von Selektionen (F=Bedingungen)
   $\sigma_{F_1}(\sigma_{F_2}(\ldots(\sigma_{F_n}(R))\ldots)) \Leftrightarrow \sigma_{F_1 \wedge F_2 \wedge \ldots \wedge F_n}(R)$
6. Kommutativität von Selektionen
   $\sigma_{F_1}(\sigma_{F_2}(R)) \Leftrightarrow \sigma_{F_2}(\sigma_{F_1}(R))$
7. Folge von Projektionen
   $\pi_{X_1}(\pi_{X_2}(\ldots(\pi_{X_n}(R))\ldots)) \Leftrightarrow \pi_{X_1}(R)$

8. Kommutativität von Selektionen und Projektionen
   - Falls Bedingung nur auf Attributmenge A:
     $\pi_A(\sigma_F(R)) \Leftrightarrow \sigma_F(\pi_A(R))$
   - Allgemeiner, wenn sich F noch auf Attribut B bezieht:
     $\pi_A(\sigma_F(R)) \Leftrightarrow \pi_A(\sigma_F(\pi_{A \cup B}(R)))$

9. Distributivität einer Selektion mit einem kartesischen Produkt oder einem natürlichen Join
   - Falls Attribute von F nur aus R, aber nicht aus S:
     $\sigma_F(R \Diamond S) \Leftrightarrow \sigma_F(R) \Diamond S$
   - Falls F_1 nur Attribute aus R, F_2 nur Attribute aus S sind und $F_1 \wedge F_2 = F$:
     $\sigma_F(R \Diamond S) \Leftrightarrow \sigma_{F_1}(R) \Diamond \sigma_{F_2}(S)$
   - Falls F_1 nur Attribute aus R, F_2 aber Attribute aus **R und S** enthält:
     $\sigma_F(R \Diamond S) \Leftrightarrow \sigma_{F_2}(\sigma_{F_1}(R) \Diamond S)$
   - F_2 nur Attribute aus R, F_3 nur Attribute aus S, $F = F_1 \wedge F_2 \wedge F_3$
     $\sigma_F(R \Diamond S) \Leftrightarrow \sigma_{F_1 \wedge F_2 \wedge F_3}(R \Diamond S) \Leftrightarrow \sigma_{F_1}(\sigma_{F_2}(\sigma_{F_3}(R \Diamond S)))$
     $\Leftrightarrow \sigma_{F_1}(\sigma_{F_2}(R) \Diamond \sigma_{F_3}(S))$

10. Distributivität einer Projektion über einem kartesischen Produkt, Durchschnitt oder Differenz:
    $\sigma_F(R \Diamond S) \Leftrightarrow \sigma_F(R) \Diamond \sigma_F(S)$
11. Distributivität einer Projektion über einer Vereinigung, einem Durchschnitt oder einer Differenz:
    $\pi_A(R \Diamond S) \Leftrightarrow \pi_A(R) \Diamond \pi_A(S)$
12. Distributivität einer Projektion über einem kartesischen Produkt oder einem Join:
    Seien B_x Attribute aus R, C_x Attribute aus S:
    - Wenn F nur Attribute aus A_x enthält:
      $\pi_{A_1, \ldots, A_n}(R \Diamond S) \Leftrightarrow \pi_{B_1, \ldots, B_k}(R) \Diamond \pi_{C_1, \ldots, C_l}(S)$
    - Wenn F noch zusätzliche Attribute enthält, die nicht in A_x sind
      $\pi_{A_1, \ldots, A_n}(R \bowtie_F S) \Leftrightarrow \pi_{A_1, \ldots, A_n}(\pi_{B_1, \ldots, B_k, B_{k+1}, \ldots, B_{k+p}}(R) \bowtie_F \pi_{C_1, \ldots, C_l, C_{l+1}, \ldots, C_{l+q}}(S))$

13. Vereinigung, Durchschnitt und Differenz auf identischen Eingaberelationen:
    $R \cup R \Leftrightarrow RR \cap R \Leftrightarrow RR - R \Leftrightarrow \varnothing$
14. Distributivität einer Selektion über einer Vereinigung, Durchschnitt oder einer Differenz auf identischen Eingaberelationen:
    $\sigma_{F_1 \vee F_2}(R) \Leftrightarrow \sigma_{F_1}(R) \cup \sigma_{F_2}(R) \sigma_{F_1 \wedge F_2}(R) \Leftrightarrow \sigma_{F_1}(R) \cap \sigma_{F_2}(R)$

    $\sigma_{F_1 \wedge \neg F_2}(R) \Leftrightarrow \sigma_{F_1}(R) - \sigma_{F_2}(R)$

Algorithmusschritte:
Siehe S. 157 ff + Selbsttestaufgabe.

5.3.2 Auswertungsplanerzeugung:

Der durch die Anfrageumformung erhaltene rel. Algebraausdruck bzw Operatorbaum muss jetzt in effizienten Auswertungsplan umgesetzt werden->den logischen Operatoren werden ausführbare Operatoren zugeordnet (=physische Operatoren) und Auswahl von Methoden zur Realisierung der ausführbaren Operatoren.
Nach der algebraischen Optimierung erfolgt Gruppierung:= Zuordnung von benachbarten Operatoren des relationalen Ausdrucks zu neuen logischen Operatoren, denen effiziente Operatoren zugeordnet werden können. ZB lassen sich Selektionen und Projektionen mit einer vorangehenden binären Operation (Vereinigung, kartesisches Produkt, Join, Differenz) gruppieren, so dass keine Zwischenrelationen mehr erzeugt werden müssen, sondern nur noch die Ergebnisrelation (=Stromverarbeitung). Beispiel:
$\pi_A(\sigma_F(R \times S))$.
Ungruppiert betrachtet: Erst kart. Produkt, dann Selektion, dann Projektion ausführen (2 Zwischenrelationen).
Gruppiert betrachtet: Vorstellung der Realisierung eines ausführbaren Operators, der sich je ein Tupel R und S nimmt, beide konkateniert, die Bedingung F prüft, bei nicht Erfülltheit das konkatenierte Tupel wegwirft und

bei Erfülltheit die Projektion bzgl A realisiert und das Ergebnis wegschreibt.
Gruppierung auch möglich, wenn eine binäre Operation Selektionen oder Projektionen als Operanden besitzt, die auf Blätter des Operandenbaumes (=Basisrelationen) angewendet werden.

- Binäroperation=Vereinigung=>Selektionen und Projektionen unterhalb im Operatorbaum können ohne Effizienzverlust hinzugruppiert werden, da die Operanden zur Vereinigungsbildung sowieso kopiert werden müssen.Unterscheidung:
- Binäroperation=kartesisches Produkt ohne nachfolgende Selektion (=Theta-Join)=>Auswertung von Selektionen und Projektionen unter Erzeugung von Zwischenrelationen, denn Größe der Operandenrelationen hat entscheidenden Einfluß auf die Laufzeit des kart. Produkts->diese Größe ist zu minimieren.

Den Gruppen sind danach ausführbare Operatoren zuzuordnen, Implementierungsalternativen im nächsten Abschnitt. Die Auswertungsplanerzeugung hat dann die Aufgabe, mit Hilfe des Angebots an Implementierungen alternative Auswertungspläne zu erzeugen, was systematisch erfolgen sollte, um die wenigen guten herauszufiltern.Dann Kostenabschätzung.
Suchstrategie legt die Reihenfolge für die Erzeugung von Auswertungsplänen fest:

- Erschöpfende Suchstrategie: Kombination aller möglichen, ausführbaren Operatoren miteinander und Erzeugung der entsprechenden Auswertungsplänen->gesamter Suchraum wird abgearbeitet, Kosten abgeschätzt und günstigster ausgewählt.
- Beschränkt-erschöpfende Suchstrategie: Gezielte Beschränkung der zu betrachtenden Kombinationen durch Parametrisierung der Auswertungsplanerzeugung, die dann nur bestimmte Typen von Auswertungsplänen in Betracht zieht (zB bestimmteJoin-Reihenfolgen).

ZB DBS System R: Eingabe für Optimierer=Operatorbaum für relationalen Ausdruck. Kandidatenpläne werden erhalten durch Permutation der Join-Reihenfolgen der n Relationen der Anfrage mittels der Kommutativitäts- und Assoziativregeln für Algebrausdrücke=>n! Kandidatenpläne.
Beschränkung des Suchraumes: Für kommutativ äquivalente Joins wird nur die billigere Kombination weiterverfolgt, kartesische Produkte werden von vorneherein eliminiert.
Grundlegende Heuristik: nur linksassoziative Join-Reihenfolgen bearbeiten wegen dessen Fähigkeit zur Stromverarbeitung. Innere Relationen (=rechter Sohn) eines Join-Knotens müssen persistent gespeichert werden, da die komplette innere Relation für jedes Tupel der äußeren Relation durchlaufen werden muss.
Vorgehensweise des Optimierers:

1. Jede Relation R mit darauf anwendbaren Selektionsbedingungen betrachten: Attribute von R, die weder in einer Selektionsbedingung noch als Projektionsattribut vorkommen, werden herausprojiziert
2. Erzeugung aller Pläne mit zwei Relationen für einen Join, indem jede aus 1. erhaltene Relation R als äußere Relation betrachtet wird und nachfolgend jede andere Relation als innere Relation S.
   - Selektionen, die sich nur auf Attribute von S beziehen, können vor dem Join ausgeführt werden.
   - Selektionen, die zur Definition des Joins dienen, werden während des Joins mittels Stromverarbeitung ausgeführt.
   - Selektionen, die sich auf Attribute anderer Relationen beziehen, können nur nach dem Join angewendet werden.
3. Alle Pläne mit 3 Relationen werden erzeugt- wie im 2.Schritt, aber nur Ergebnisse des 2.Schritts werden als äußere Relation verwendet.

Iteration dieser Vorgehensweise in weiteren Schritten, bis Auswertungspläne erzeugt werden, die alle n Relationen in der Anfrage enthalten->Nach dem n-ten Schritt kann billigster Auswertungsplan bestimmt werden.
Kosten eines Kandidatenplans= gewichtete Kombination aus Kosten für I/O-Operationen und CPU-Kosten. Schätzung zur Übersetzungszeit durch Kostenmodell, das eine Kostenformel für jede Low-level-Operation (zB Suche mit Hilfe eines B+-Index bzgl eines Bereichprädikats) bereithält.
Weitere Suchstrategien: Zufallsgesteuerte Suchstrategien. Umfassen generische Algorithmen,die einen Graph durchsuchen, dessen Knoten alle alternativen Auswertungspläne darstellen, die zur Beantwortung einer Anfrage verwendet werden können. Kanten entsprechen Transformationen des Plans. Charakteristisch für diese Algorithmen: Wandern mittels einer Folge von Bewegungen zufallsgesteuert durch den Graphen.

- Wiederholte Verbesserung: Führt große Anzahl von lokalen Optimierungen durch. Jede startet an einem zufälligen Knoten, wiederholte Abwärtsbewegungen bis lokales Minimum erreicht ist. Rückgabe des kleinsten lokalen Minimums
- Simuliertes Härten: Akzeptiert mit gewisser Wahrscheinlichkeit auch Aufwärtsbewegungen (höhere Kosten), um nicht in Knoten lokalen Minimums festzusitzen. Diese Wahrscheinlichkeit nimmt mit zunehmender Zeit ab und wird schliesslich null.Wahrscheinlichkeitswert wird bestimmt durch eine Funktion auf Basis einer Exponentialfunktion, die den Effekt des Aushärtens (zB von Metallen) simuliert.

- Zwei-Phasen-Optimierung: Kombination aus beiden: Zuerst Wiederholte Verbesserung->bestes lokales Minimum ist Anfangsknoten des Simulierten Härtens.

5.3.3 Implementierung relationaler Operatoren

Unterscheidung Operatoren und Strategien, die den Zugriff auf die Tupel **einer oder mehrerer** Relationen erlauben.

- **Zugriff auf alle Tupel einer Relation**
  Immer mögliche Standardmethode: Relationendurchlauf!
  Alle Seiten einer Relation werden nacheinander gelesen und die gesuchten Tupel herauskopiert.
  Nachteil: Hohe Kosten.
  Effizientere Methode: Verwendung von Zugriffspfaden (Index-, Hash-Strukturen)-> Indexdurchlauf liest nacheinander alle Indexseiten und stellt die TIDs der gesuchten Tupel in Sortierreihenfolge der Indexstruktur bereit. Werden die TIDs zusätzlich vorab sortiert oder ist die Relation geclustert, muss jede Seite der Relation höchstens einmal gelesen werden.
- **Selektion**
  - kein Index und unsortierte Daten: Relationendurchlauf nötig->O(n) Seitenzugriffe
  - Daten sortiert bzgl des Selektionsattributs->binäre Suche möglich->O(log n) Zugriffe
  - Index liegt vor:
    - Ist der Selektionsoperator ein Gleichheitsprädikat->Hash-Index ist am geeignetsten mit 1-2 Seitenzugriffen, B+-Baum mit 2-3 Seitenzugriffen.
      - Selektionsattribut=Primärschlüssel->höchstens ein passendes Tupel
      - Selektionsattribut=Sekundärschlüssel->mehrere Tupel
      - Sekundärindex=Primärorganisation/geclustert->zusätzlich einige weitere Zugriffe
      - Sekundärindex=Sekundärorganisation+nicht geclustert->jeder Indexeintrag kann auf andere Seite verweisen->zusätzliche Zugriffe gemäß Zahl der passenden Tupel
    - Selektionsoperator=Vergleich (<,<=,>,>=)->Bereichsanfrage->TID-basierter Index (B+-Baum geeignet)->Suche des ersten Indexeintrags im Blatt, das Bedingung erfüllt->dann alle verketteten Blattseiten des Index solange vorwärts bzw rückwärts durchlaufen, bis Bedingung nicht mehr zutrifft.
  - Selektionsbedingung in konjunktiver Normalform:
    - Index auf einfachem Attribut, für das Index existiert->Tupel ermitteln und auf andere Bedingungen prüfen.
    - Zusammengesetzter Index auf mehreren Attributen mit Gleichheitsbedingung: Sorfort verwendbar.
    - Index auf mehreren Attributen: Sammeln der TIDs für jedes Attribut und Verschneiden miteinander, dann auf übrige Bedingungen prüfen.
  - Selektionsbedingung in disjunktiver Normalform:
    - wenn auch nur eine Bedingung keinen Zugriffspfad besitzt->lineare Suche
    - jede Bedingung hat Zugriffspfad->für jede Bedingung die passenden Tupel ermitteln, und anschliessende Vereinigung unter Duplikateleminierung->externes Sortieren nötig. Besser: für jede Bedingung die TIDs ermitteln und vereinigen.
- **Projektion**
- **Join**
- **Mengenoperationen**

5.3.4 Kostenschätzung

Zur Auswahl des billigsten Auswertungsplans wird jedem ausführbaren Operator eine Kostenfunktion zugeordnet:=Schätzung. Vorausgesetzt werden gleichmäßige Verteilung der Werte eines Attributs und Unabhängigkeit der Werte verschiedener Attribute einer Relation.
Kostenunterteilung in 1. Zugriffskosten (extern) 2. Berechnungskosten 3. Kommunikationskosten.
Im weiteren nur Zugriffskosten:
Im Systemkatalog werden ergänzend zu den qualititiven Datenbankbeschreibungen Statistiken zur quantitativen Beschreibung der Daten gehalten, zB Zahl der Tupel einer Relation, Zahl der Blöcke einer Relation, Blockungsfaktor, Sortierung...
Kostenabschätzung eine Selektion: Die Kardinalität des Ergebnisses einer Operation ist die Selektivität der Selektion mal der Kardinalität der Relation (Selektivität = Anteil der erfüllenden Tupel zu allen Tupeln von R):

$$card(\sigma_F(R)) = s_\sigma(F, R) \cdot card(R)$$

Exakte Schätzung von Selektivitäten schwierig. Unter obiger Annahme der Gleichverteilung kann die Selektivität aber so berechnet werden:

$$s_\sigma(A = v, R) = 1 \,/\, card(\pi_A(R))$$

$$s_\sigma(A = v, R) = 1 \,/\, card(I_R(A))$$

$$s_\sigma(A > v, R) = (max(A) - v) \,/\, (max(A) - min(A))$$

$$s_\sigma(A < v, R) = (v - min(A)) \,/\, (max(A) - min(A))$$

$$s_\sigma(A > v_1 \wedge A < v_2, R) = (v_2 - v_1) \,/\, (max(A) - min(A))$$

$$s_\sigma(p(A) \wedge p(B), R) = s_\sigma(p(A), R) \cdot s_\sigma(p(B), R)$$

$$s_\sigma(p(A) \vee p(B), R) = s_\sigma(p(A), R) + s_\sigma(p(B), R) - s_\sigma(p(A), R) \cdot s_\sigma(p(B), R)$$

$$s_\sigma(\neg p(A), R) = 1 - s_\sigma(p(A), R)$$

$$s_\sigma(A \in V, R) = s_\sigma(A = v, R) \cdot card(V) = card(V) \,/\, card(\pi_A(R))$$

Im ersten Fall $s_\sigma(A=v, R)$ ergeben sich: card(R)/V(A,R) Tupel bzw einer, wenn A Primärschlüssel ist (denn dann ist card(R)/V(A,R)). Die Selektivität $s_\sigma(A=v, R)$ ist dann gleich 1/card(R).

Kostenabschätzung eines Joins: Die Join-Selektivität ist der Anteil der Tupel, die am Join zweier Relationen R und S teilnehmen. Schlechte Join-Selektivitäten haben einen Wert von 0,5 (große verbundene Relationen), Werte von 0,001 sind gute Selektivitäten.

$$s_{\bowtie}(F, R, S) = card(R \bowtie_F S) \,/\, (card(R) \cdot card(S))$$

- Ist die Join-Bedingung A=B und sind A,B identische oder kompatible Attribute:
$$s_{\bowtie}(F, R, S) = card(\pi_B(S)) \,/\, card(dom(A))$$
- Liegt auf **A und B** ein Index:
$$s_{\bowtie}(A = B, R, S) = 1 \,/\, max(I_R(A), I_S(B))$$
- Liegt auf **A oder B** ein Index:
$$s_{\bowtie}(A = B, R, S) = 1 \,/\, I_R(A) \text{ bzw. } s_{\bowtie}(A = B, R, S) = 1 \,/\, I_S(B)$$

Die obere Schranke der Kardinalität eines Joins ist die des kartesischen Produkts:

$$card(R \times S) = card(R) \cdot card(S)$$

Manchmal wird dieser pessimistische Wert genommen, manchmal dieser durch eine Konstante geteilt. Einfach in folgendem Fall:

- Ist A eine Schlüsselattribut und B ein Fremdschlüsselattribut :
$$card(R \bowtie_{A=B} S) = card(R)$$

Für andere Joins lohnt es, die Join-Selektivität zu protkollieren, die Kardinalität des Ergebnisses ist dann:

$$card(R \bowtie_F S) = s_{\bowtie}(F, R, S) \cdot card(R) \cdot card(S)$$

5.4 Anfrageausführung

Anfrage-Ausführer berechnet zur Laufzeit das Anfrage-Ergebnis.
Ist die Anfrage parametrisiert->zwei Strategien:

- Eingabe wird direkt interpretiert (Operatoren werden nacheinander ausgeführt)->Anfrage-Ausführer ist dann normales Interpretiersystem
- Wiederholte Anfragen: Durchführung einer Code-Generierung und diesen Code entsprechend oft ausführen.

Allgemeine Verarbeitungsstrategie eines ausführbaren Operators: **Open-next-close-Protokoll**:=open-Funktion initialisiert den Operator, next übergibt das nächste Ergebnisobjekt, close beschließt die Verarbeitung.
Vorteil: Operatoren können leicht lokal gegeneinander ausgetauscht werden, solange sie semantisch äquivalent sind.
Die Art des Lesens eines Eingabestroms bzw Schreiben eines Ausgabestroms von Objekten bleibt dem Operator überlassen (Ausgabe sortiert, indiziert, externe Zwischenspeicherung etc mengenorientierte Strategie

[alle Ergebnisobjekte auf einmal erzeugen]/tupelorientierte Strategie [nach Schreiben jedes Ausgabeobjekts Kontrolle an Nachfolgeoperator abgeben]).
Erzeugter Code für einen Auswertungsplan wird meist mit der ursprünglichen Anfrage im Systemkatalog abgelegt und dazu die Beziehungen zu anderen Meta-Daten (Speicherungsstrukturen und Zugriffspfade)-diese Beziehungen beschreiben die Abhängigkeiten eines Auswertungsplans zu seiner konkreten Datenbankumgebung. Der Anfrage-Ausführer entscheidet jedesmal vor Ausführung eines Auswertungsplans durch die gespeicherten Beziehungsdaten, ob bei Änderungen eine Neuübersetzung/Reoptimierung nötig ist.