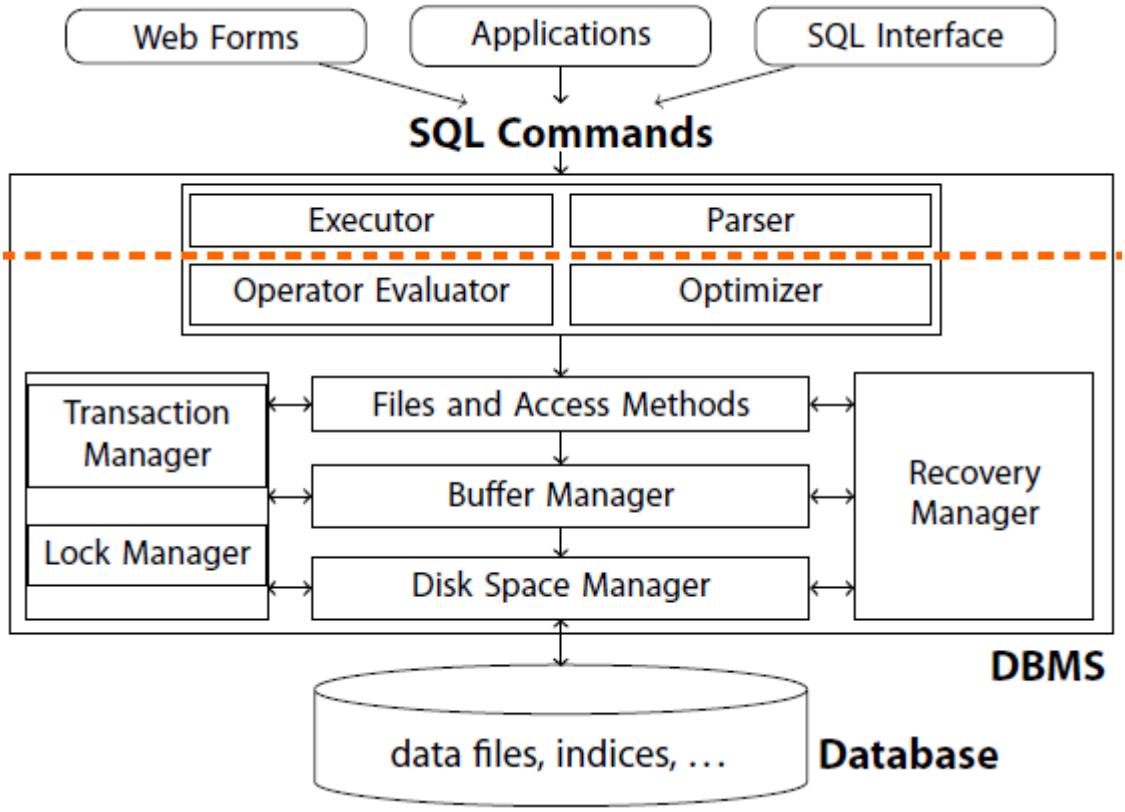
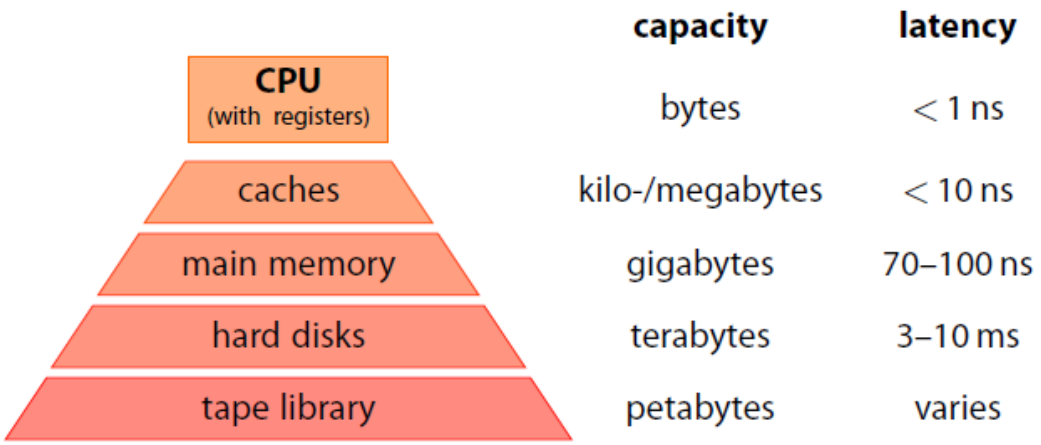
Chapter 1 – Introduction

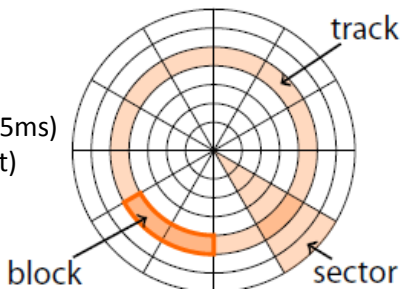


Chapter 2 – Storage

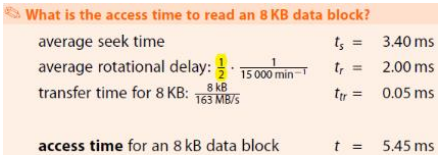


- Mittels Cache wird die Latenz versucht zu verstecken.

2.1 Magnetic Disks



- **Access Time** setzt sich aus dem Folgendem zusammen: (z.B. 5,45ms)
  - seek time (Zeit bis der Arm über dem richtigen ‘track’ ist)
  - rotational delay
  - transfer time



- **Sequentiell**es Lesen ermöglicht deutliche Beschleunigung, da die ‚seek time' wesentlich seltener auftritt und das ‚rotational delay' nur einmal. (16 Tracks benötigt, 1000 Blocks gelesen) $t_{seq} = t_s + t_r + 1000 * t_{tr} + 16 * t_{s,track-to-track}$
- **Random** I/O kostet unverhältnismäßig viel und wird daher versucht zu vermeiden
- Sobald ca. 1% (bei Seagate Cheetah 15K.7) gelesen werden, lohnt es sich die Datei Sequentiell zu lesen.
- Performance Tricks:
  - **track skewing**: Sektor 0 ausrichten um rotational delay zu vermeiden
  - **request scheduling**: request mit kürzestem Weg des Arms wird zuerst bedient
  - **zoning**: äußere Tracks werden in mehrere Sektoren unterteilt (da Track länger)
- seek time/rotational delay verbesserten sich kaum gegenüber der transfer time über die Jahre, daher ist random access heute teurer als früher

2.2 I/O Parallelism

- mittels Parallelisierung wird die Latenzzeit umgangen (mehrere Disks)
- RAID1 (spiegeln)
- RAID0 (Inhalt auf mehrere Platten verteilen, **striping**)
- RAID5 (striping mit Parität, ab 3 Platten, 1 Ausfall möglich)
  RAID6 (wie RAID5, zwei Ausfälle möglich)

Example (Read 1 000 blocks of size 8 kB)

- **random access:**
  $t_{rnd} = 1\,000 \cdot 0.30\,\text{ms} = 0.3\,\text{s}$
- **sequential read of adjacent pages:**
  $t_{seq} = \left\lceil \frac{1\,000 \cdot 8\,\text{kB}}{128\,\text{kB}} \right\rceil \cdot t_{tr} \approx 18.9\,\text{ms}$

  The Seagate Pulsar.2 (sequentially) reads data in 128 kB chunks.

2.3 Alternative Storage Techniques

2.3.1 Solid-State Disks

- *Vorteil*: niedrige Latenz, schnelles lesen und schreiben
- *Nachteil*: Speicherkapazität teuer, schnelle Abnutzung
- Random Access nur wegen kleinen Blocks nachteilhaft. (seek time, rot. delay entfällt)

2.3.2 Network-Based Storage

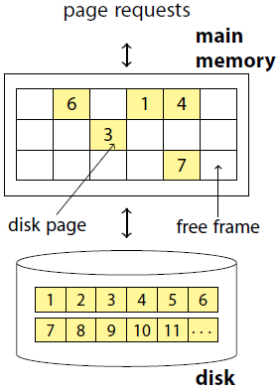
- Storage Area Network (SAN): Block basierter Zugriff auf den Speicher, abstrahiert vom RAID oder physischem Speicher (→ Hardwarebeschleunigung, vereinfachte Nutzung)
- Grid or Cloud Storage: Verwendet Cluster von vielen Standard PCs, massive Spiegelung, CPU und Speicher als Service verkauft

2.4 Managing Space

- Disk Space Manager: Abstrahiert über die Details der darunterliegenden Speicher.
- Stellt ‚Page'-Konzept als Einheit von Speicher gegenüber den übrigen Systemkomponenten dar
- Stellt Mapping zur Verfügung: page number ↦ physical location (z.B. file + offset, …)
- DSM merkt sich benützte und freie Blocks (Linked List der freien pages oder **bitmap** über alle pages ob belegt oder frei → ermöglicht fortlaufende Pages → sequentielles Lesen)

2.5 Buffer Manager

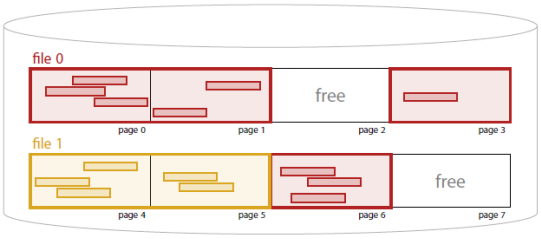
- vermittelt zwischen externen Speicher und Hauptspeicher, verwaltet den Hauptspeicher für diese Aufgabe, merkt sich welche Pages derzeit noch benötigt werden
- Mittels **Ersetzungsstragie** wird entschieden welche Pages bei vollem überschrieben werden
- Bietet **pin(pageno)** und **unpin(pageno,dirty)** für Higher-Level Code an
  (dirty gibt an ob die page verändert wurde)
- Typischerweiße keine erzwungenes zurückschreiben bei unpin in DBMS.



- Simultane Änderungen an einem Block werden mittels der **Concurrency control** verarbeitet. Der Buffer Manager nimmt an, dass alles in ordnung ist, wenn er *unpin(p,true)* erhält
- **Ersetzungsstrategien**:
  - *Least Recently Used* (LRU): Page bei dem das letzte unpin am längsten her ist
  - *LRU-k*: Wie LRU, berücksichtigt aber die letzten k unpin Aufrufe einer Page
  - *Most Recently Used* (MRU): Page überschreiben die als letztes unpined wurde
  - *Random*
  - *Second Chance*: LRU-Simulation mit wenig Overhead, bei pin(p) wird referenced auf 1 gesetzt, pages werden durchlaufen und wenn referenced auf 0 ist, dann wird überschrieben, ansonsten referenced auf 0 gesetzt und die nächste page geprüft.
- Heuristische Stragegien können fehlschlagen (z.B. LRU, mehrmaliger SELECT über Tabelle, die nicht in den Buffer passt)
- **Herrausforderungen** für die LRU: T_1 führt wiederholt über eine feste Menge Transaktionen aus, T_2 führt sequentieller Scan über die Datebankpages durch
- In der Praxis:
  - **Prefetching**: versucht Pagerequests zu überlappen und spekuliert z.B. auf sequential scan, manager bietet Funktion an mehrere Seiten vorrauszuladen
  - **Page fixing/hating**: Higher-Level Code fordert '**fix**' einer Page wenn sie in nächster Zukunft nützlich wird, entsprechend '**hate**' falls nicht in nächster Zeit benützt
  - **Partitioned buffer pools**

2.7 Databases vs. Operating Systems

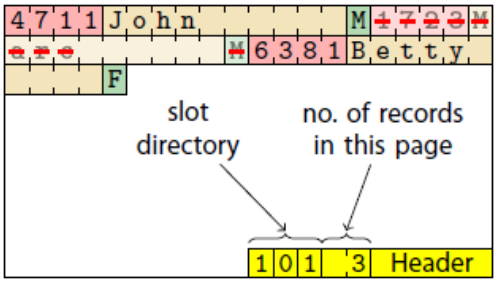
- DSM & BM sind ähnlich zum Dateimanagement und virtuellen Speicher in Betriebssystemen
- aber ein DBMS kann viel mehr Zugriffspattern für bestimmte Operatoren verwenden
- Simultane Ausführung fordert vorgeschriebene Ordnung der Schreiboperationen
- DBMS versucht Betriebssystembuffer zu umgehen und lieber auf die Rohdaten selbst zuzugreifen, da es selbst effizienter damit umgehen kann



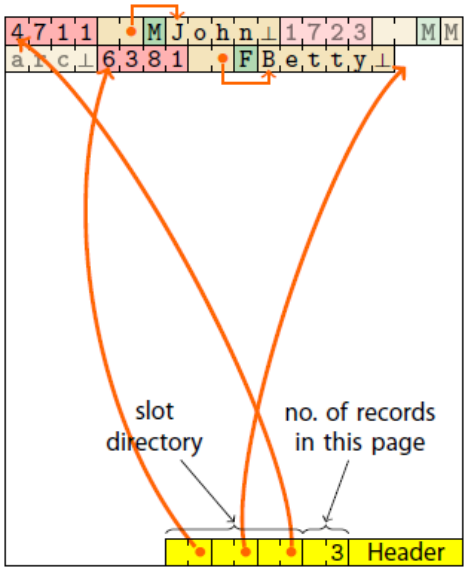
2.8 Files and Records

- Eine Datei besteht aus einer oder mehreren Pages
- Eine Page besteht aus einem oder mehreren Records
- Jedes Record entspricht einem Tupel
- typisches Heap-File Interface
  - f = createFile(n), deleteFile(n)
  - rid = insertRecord(f,record)
  - deleteRecord(f,rid)
  - record = getRecord(f,rid)
  - openScan(f) *(sequential scan)*
- Einfache Implementierung: **Linked Lists** of Pages, header page mit angehängten freien und vollen pages → uneffizient
- **Directory of pages**: Verwenden einer Speicherplatzmap mit den Informationen über den freien Speicherplatz, suche nach freiem Speicher effizient, aber großen Overhead für die Map
- Free Space Management
  Append Only (immer in letzte Page einfügen, sonst neue), **Best Fit**, **First Fit**, **Next Fit**
- Optimierung mit **Free Space Witness** (Einteilung in 25% Schritte der Füllmenge)

12. Juni 2014



- Der interne Aufbau einer Page:
  - **Fixed-Length Records**
  - Variable-Sized Fields: Variable Felder werden zum Ende des Records verschoben (konstanter Offset)
    Records können sich ‚bewegen'
    Adresse zeigt auf andere Seite, falls Record nicht mehr in die aktuelle Page passt.
  - Alternative Page Layouts: row-wise, column-wise
    - wird auch als NSM (n-ary storage model) und DSM (decomposition storage model) bezeichnet.



Chapter 3 - Indexing

3.1 File Organisation

- „heap file" unterstützt lediglich sequentielle Scans (openScan(·))

3.2 File Organization Competition

- 3 Datei Organisationssysteme
  - Dateien aus zufällig geordneten Tupeln ("heap files")
  - Dateien sortiert nach gewissen Feldern
  - Dateien hashed nach gewissen Feldern
- Eine Dateiorganisation kann immer nur **eine** Query-Klasse effizient machen.
- Mittels **Index**en können mehrere Query-Klassen effizient gemacht werden.

3.2.1 Simple Cost Model

	Beschreibung	Üblich
b	# der Seiten in einer Datei	
r	# der Records in einer Seite	
D	benötigte Zeit zum schreiben/lesen einer Plattenseite	$\approx 15ms$
C	CPU Zeit die benötigt wird (z.B. für Vergleiche eines Feldes	$\approx 0{,}1\ \mu s$
H	CPU Zeit die benötigt wird um eine Funktion anzuwenden (z.B. Hashfunktion)	$\approx 0{,}1\mu s$

- Die Hashfunktion bestimmt die ID der Seiten, der Platz darin wird nicht beschrieben dadurch
- Eine Seite ist kann „**overflow**" Seiten erhalten, falls sie voll ist, um diese zu vermeiden, sind Seiten üblicherweise zu 80% gefüllt, wenn die Heapdatei initial gehasht wird.

3.2.2 Scan

- **Heap Datei**: Alle Seiten lesen, sowie alle Records auf jeder Seite zugreifen
$$Scan_{heap} = b \cdot (D + r \cdot C)$$
- **Sortierte Datei**: (wie Heap Datei)
$$Scan_{sort} = b \cdot (D + r \cdot C)$$
- **Gehashte Datei**: im Prinzip wie Heap Datei, nur dass hier der freie Speicherplatz übersprungen werden kann (siehe 3.2.1)
$$Scan_{hash} = \frac{100}{80} \cdot b \cdot (D + r \cdot C)$$

3.2.3 Equality Test (A = const)

- a) Gleichheitstest auf dem Primärschlüssel, b) Gleichheitstest nicht auf dem Primärschlüssel
- **Heap Datei**:
  a. $Search_{heap} = \frac{1}{2} \cdot b \cdot \big(D + r \cdot (C + H)\big)$

b. $Search_{heap} = b \cdot \left(D + r \cdot (C + H)\right)$

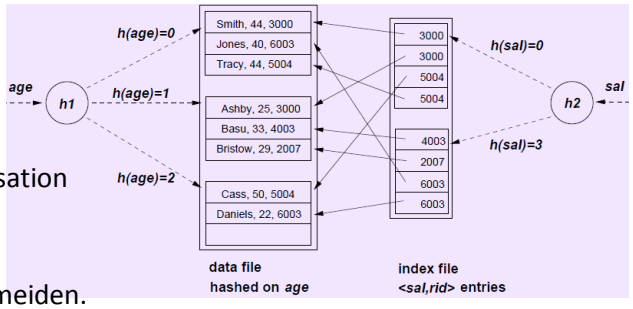
- **Sortierte Datei**: (sortiert nach A)
  Angenommen wir verwenden Binäre Suche (was tatsächlich aufgrund der unvorhersehbaren Sprünge kein DBMS implementieren wird)
  $$Search_{sort} = \log_2 b \cdot (D + \log_2 r \cdot (C + H))$$
- **Gehashte Datei**: (gehasht nach A) Bietet die beste Unterstützung! („Overflow chain“ ignoriert)
  a. $Search_{hash} = H + D + \frac{1}{2} \cdot r \cdot C$
  b. $Search_{hash} = H + D + r \cdot C$

3.2.4 Range Selection ($A \leq lower\ AND\ A \leq upper$)

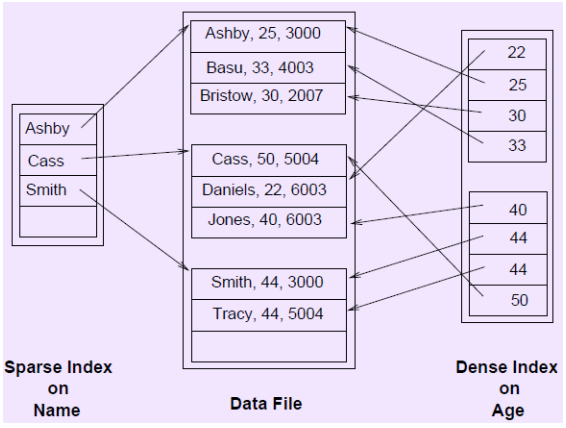
- Heap Datei:
- Sortierte Datei (sortiert nach A):
- Gehashte Datei (gehasht nach A):

3.3 Indexes

- Mit Index Strukturen (B+-Bäume) können alle Vorteile von sortierten Dateien erreicht werden und effizient neue Tupel eingefügt und gelöscht werden.
- Mittels einer Index-Struktur kann nachträglich eine benötigte Unterstützung erreicht werden.
- Index entry designs
  a. <k, <…, A = k, …>>
  b. <k, rid>
  c. <k, [rid_1, rid_2,…]>



- Bei a. ist der Index selbst eine spezielle Dateiorganisation
  → Records nicht extra speichern
- Verwendet man mehrere Indexe, sollte wenigstens einer Variante a. verwenden um Redundanz zu vermeiden.
- b. und c. verwenden Zeiger, um auf die tatsächlichen Datenfiles zu zeigen.
- c. führt zu weniger Index-Einträgen, aber dafür variablen Längen.
- **Clustered index**: Ein Index nach dem die Daten Datei sortiert wird.
- Clustered Index effizienter für Range Selection, unclustered Range Selection ist uneffizient für und die Kosten wachsen mit der Größe der Range stark
- In DB2 Clustered Index erstellen: CREATE INDEX IXR ON R(A ASC) CLUSTER
  In Postgres Clustered Index erstellen: CLUSTER R USING IXR (Index IXR muss schon bestehn)
- **Sparse Index**: Um die Größe des Indexes klein zu halten, wird nur ein Zeiger pro Daten Page angelegt. Der Schlüssel k ist der kleinste Schüssel der Page. Ein Sparse Index ist immer clustered!
- **Dense Index**: Hier wird für jedes Record ein Zeiger angelegt.

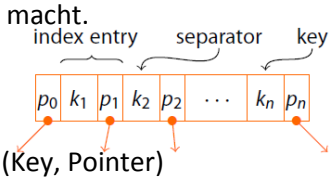


Chapter 4 – Tree Structured Indexing

4.1 Binary Search

- Binäre Suche macht unvorhersehbare Sprünge, was das Vorladen unmöglich macht.

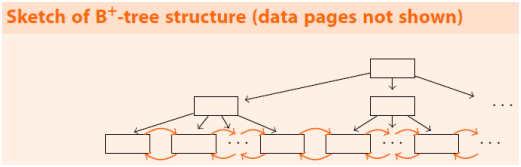
4.2 ISAM



- ISAM ist der Ersatz für die Binäre Suche und benötigt weniger Seiten dazu.
- Zusätzlich zur sortierten Daten Datei wird noch eine Index Datei verwendet. (Key, Pointer)
- ISAM führt zu Sparse Index Strukturen. (Ein Pointer auf den ersten Eintrag einer Page)
- Es wird garantiert, dass $k_{i-1} < k_i$
- Range Selection mittels Binärer Suche auf der Indexdatei. (Vorteil: wenige I/Os)
- Multi-Level ISAM: Bei großen Datenmengen werden Rekursiv mehrere ISAM Indexe angewendet, bis der oberste ISAM Index in eine Page passt.
- ISAM Struktur mittels bottum-up Methode erstellen. Üblicherweiße werden 20% frei gelassen, um spätere Änderungen besser aufzufangen.
- ISAM Index bleibt statisch gleich, mit vielen Änderungen werden **Overflow Page**s benötigt und die Performance wird abgeschwächt.
- Dank statischem Index erspart man sich die Locks bei Concurrent Index Zugriff. Dies ist bei dynamischem Index nicht möglich.
- Trotz genannter Defizite ist ISAM-basiertes suchen die effizenteste geordnete Indexsuche die bis jetzt diskutiert wurde.
- ISAM fanout: Pro ISAM-Level wird der Suchraum auf die Größe $N \cdot \frac{1}{F}$ reduziert.
  (F ist die maximale Anzahl an Kinder pro Index Knoten, N die Anzahl der Seiten)
- Beispiel: Bei 10^9 Pages benötigt ISAM nur 3 Pages.

4.3 B⁺-trees

- Such Performance hängt nur von der Höhe ab, welche aufgrund hohem fanout gering ist
- Keine Overflow Pages wie bei ISAM → balance bleibt erhalten
- Daten Dateien kann sich dynamisch verändern
- Blätter enthalten die Daten oder nur die Referenzen zu den Records (üblicherweiße letzteres)

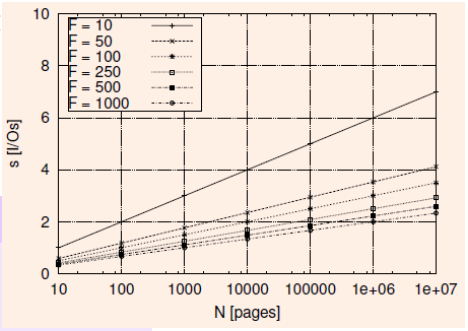


- Die minimale und maximale Anzahl an Einträgen hängt von der Ordnung d ab. (Wurzel ausgenommen)
  $d \leq n \leq 2 \cdot d$ (root node: $1 \leq n \leq 2 \cdot d$)
- Ein Blatt-Knoten enthält $n + 1$ Zeiger, $n - 1$ davon zu den Records und einer auf das vorherige sowie einer auf nächste für sequentielles auslesen.
- Schlüsselwerte in B⁺-Bäumen sind eindeutig.
- Einfügen in B⁺-Bäume:
  - Blatt suchen in die der Eintrag <k,q> hingehören würde
  - Wenn genügend Platz vorhanden einfach dort einfügen.
    Ansonsten wird Knoten p geteilt in p und p' und p' entsprechend im Elternknoten eingefügt. → rekursives Vorgehen
    (Beachte: Unterscheidung zwischen Blattknoten bei denen ein Eintrag kopiert wird und inneren Knoten bei denen ein Eintrag für die Rekursion entfernt wird)

h	# records
2	16,000
3	2,000,000
4	250,000,000

- Umverteilung: Um die durchschnittliche Belegung eines B⁺-Baumes zu erhöhen.
  Beim Einfügen werden die Geschwister betrachtet und dort eingefügt, falls es noch Platz hat.
  Die Elternknoten müssen hierbei aktualisiert werden. → erspart unnötige Splits

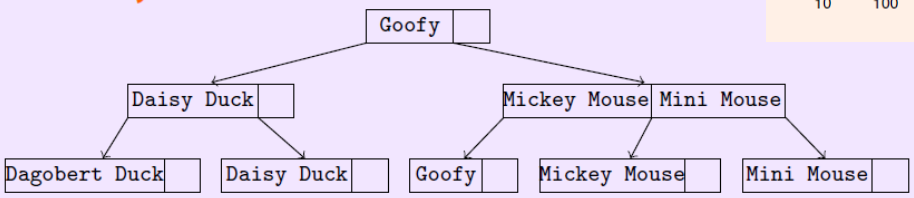
- Löschen in B⁺-Bäumen:
  - Mittels Suche das Blatt suchen mit dem entsprechenden Record.
  - Wenn die Anzahl der Einträge in diesem Blattknoten $m \leq d$: wird der link Geschwisterknoten vereint und entsprechend die Pointer angepasst
- DBMS vermeiden oft die Kosten eines Merges und sehen die Minimumregel entspannter.
- Um mit doppelten Schlüsseln umzugehen bildet DB2 intern einen Schlüssel <k, rid> Alternative: Beachten von Duplikaten in der Struktur (k* = <k, [rid1, rid2, …]>) oder behandeln von Duplikaten wie alle anderen Schlüssel. Hier muss allerdings dann die Suche leicht abgeändert werden, sodass entsprechend auch alle Treffer zurückgegeben werden.
- Es zahlt sich aus, den Fanout F zu maximieren, um die I/Os gering zu halten. $s = \log_F N$



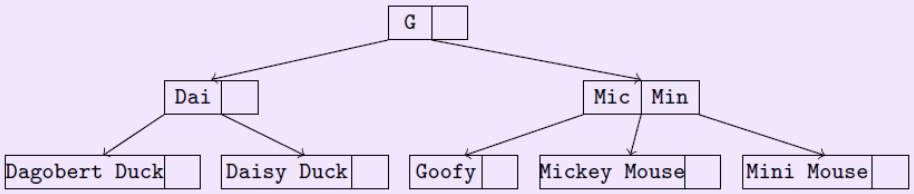
- In den inneren Knoten reicht es aus d die Schlüssel so gering wie möglich zu halten, es wird nur so viel benötigt, um diese als Wegweiser benützt zu können. → **Schlüsselkomprimierung**

Example (Key suffix truncation, B⁺-tree with order $d = 1$)

Before key suffix truncation:

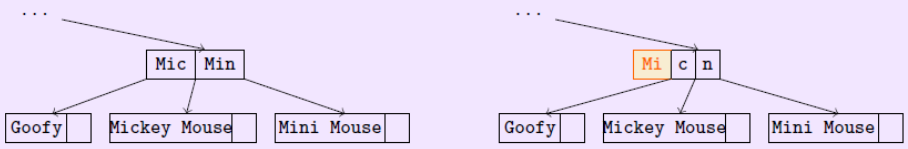


After key suffix truncation:



Durch verletzen der 50% Belegungsregel kann die Präfixkompression verbessern:

Example (Shared key prefixes in inner B⁺-tree nodes)



- **Bulk Loading**: Wird verwendet, wenn in einen Baum viele Tupel eingefügt werden müssen. Die Tupel werden dabei nach dem Schlüssel des Baumes sortiert und der Baum wird von den Knoten aus aufgebaut. → erspart unnötige Splits
- **Paritioned B-Trees** werden verwendet, um schnell Aussagen über Felder des Schüssels zutreffen.

Chapter 5 – Multi-Dimensional Indexing

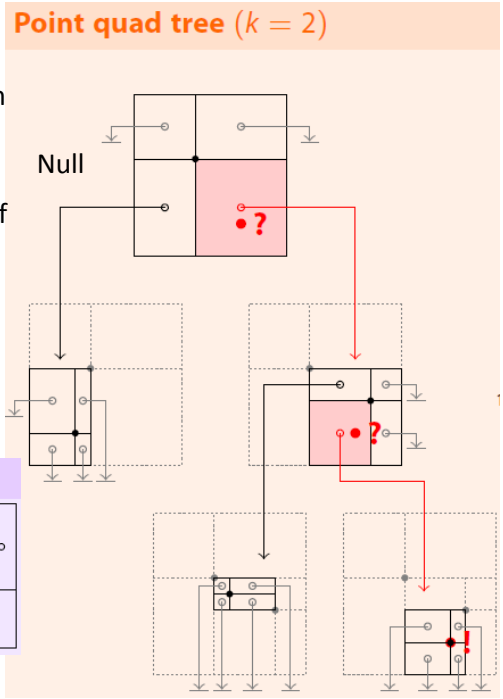
- Typischer Beispiele für Multi-Dimensionale Daten beinhalten
  - Georgaphische Daten
  - Multimedia (Bilder, Video)
  - OLAP (Online Analytical Processing)
  - Abfragen gegen codierte Daten (z.B: XML)
- Jede Gleichheitssuche kann auf eindimensionale Queries reduziert werden.
- Es gibt für Multidimensional Daten verschiedene Ansätze, wovon keiner Allgemein überzeugt.

5.1 B⁺-trees

- Können 2 B⁺-Bäume einen 2-Dimsionalen Range-Query unterstützen?
  Nein! Viele falsche Treffer müssen betrachtet warden!
- Zusammengesetzte Schlüssel helfen nicht, dieses Problem zu lösen.
- Reine B⁺-Bäume können nur ein dimensionale Querys effizient beantworten.

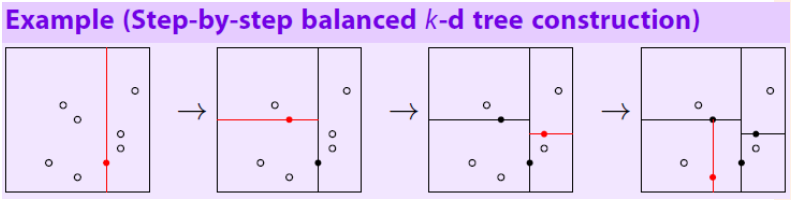
5.2 Point Quad Trees

- In k Dimensionen wird ein binärer Suchbaum ein 2^k-närer Baum
- Einfügen von Punkten:
  - Durchlaufe den Baum bis zu einem Nullzeiger und erstelle neuen Knoten, wobei alle Kindknoten wieder auf ein Nullzeiger sind. (→ keine Balance)
- Kann Regions-Abfragen beantworten



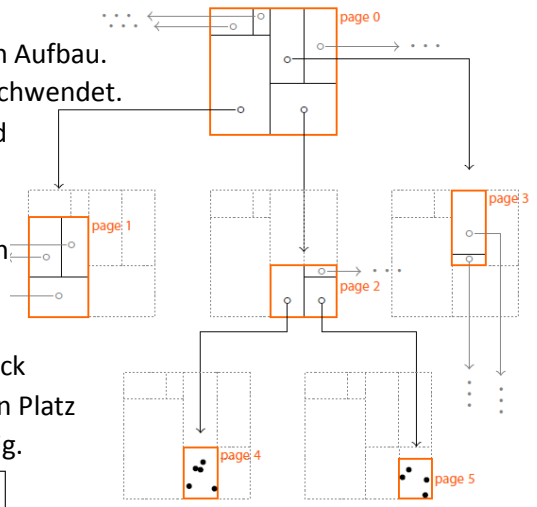
5.3 k-d Trees

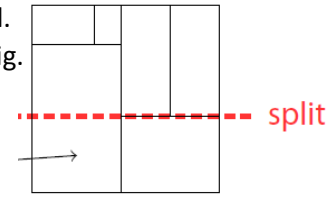
- Ähnlich zu Quad Trees nur binär. Damit wird Speicher gespart und die Vorteile von Quad Trees bleiben trotzdem erhalten.

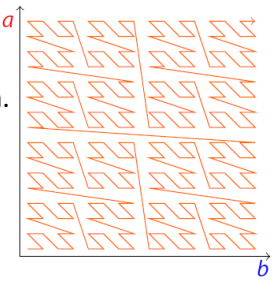


5.4 K-D-B-Trees

- K-d Bäume verbessern die Defizite von Quad Trees:
  - K-d Bäume können balanciert werden durch erneuten Aufbau.
  - Es werden nicht große Mengen an Speicherplatz verschwendet.
- Seiten werden als organisatorische Einheit verwendet und k-d Baumähnliches Layout um jede Seite zu organisieren
- Suchen in K-D-B-Bäumen:
  - In jeder Seite alle Regionen R_i bestimmen welche den Query Punkt/einen Teil der Suchregion enthalten.
  - Für jeden Treffer R_i suche rekursiv
  - Auf den Punkt-Seiten gib die passenden Records zurück
- Einfügen in K-D-B-Bäume verläuft ähnlich zu B⁺-Bäume. Sofern Platz in die Punktseite einfügen, ansonsten rekursiv teilen falls nötig. Regionseiten Splits nicht nicht trivial, eventl. Ist ein rekursiver Split nach unten notwendig.





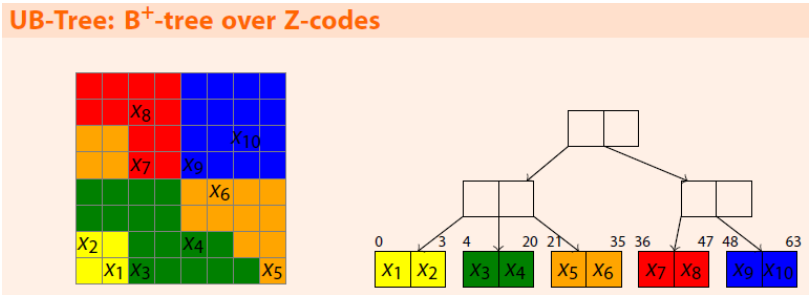


- K-D-B-Bäume (wie k-d-Bäume) können Löschungen nicht dynamisch behandeln.
- Sie sind symmetrisch in Bezug auf alle Dimensionen.

5.5 R-Trees (nicht klausurrelevant)

5.6 UB-Trees

- UB-Bäume nützen Bit Verschachtelung (Z-Ordnung) und erreichen damit ein räumliches Clustering.



- Range Query: Nachdem jede Seite beachtet wurde, wird ein index re-scan (nicht trivial) benötigt, um die nächste Blatt-Seite zu finden welche in unserer Range liegt.
- Alternativ: andere Verfahren, um die Daten zu linearisieren (z.B. Hilbert Kurven)

5.7 Spaces with High Dimensionality

- Alle Techniken die hier besprochen wurden sind bei höherem k massiv uneffizient.
- Selbst wenn unsere Suchregion 95% in jeder Dimension abdeckt, so ist die Wahrscheinlichkeit, dass ein Punkt in dieser Region nur bei: $0{,}95^k$ ➔ „curse of dimensionality“
  ➔ sequentielles Scannen im Verhältnis wieder effizient

Chapter 6 – Hash-Based Indexing

6.1 Hash Based Indexing

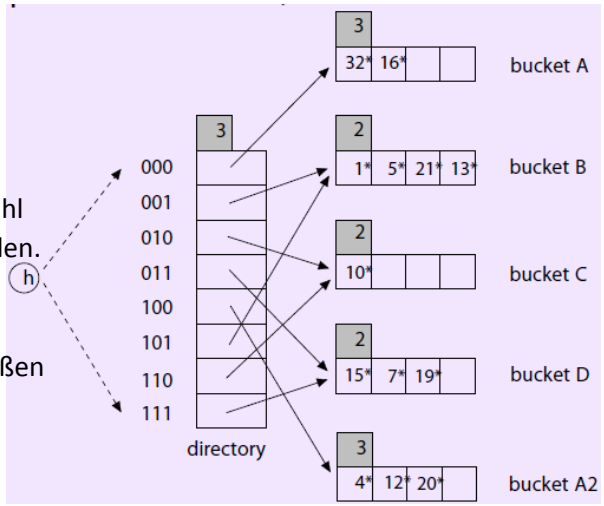
- Hash Index sind unschlagbar bei Gleichheitsabfragen, versagen aber bei Range Querys

6.2 Static Hashing

- Unterstützt keine Veränderungen (wie ISAM)
- Konstruktion:
  - Speicherplatz über N Seiten (primäre Buckets) allokieren
  - Jedes Buckets enthält einen Zeiger auf eine Liste an Overflow Seiten (zu Beginn NULL)
  - Hash function h definieren: $h: INTEGER \rightarrow [0, \dots, N-1]$
- Jedes Bucket enthält die Index Einträge und implementiert eine der 3 Varianten (siehe 3.3)
- Wenn die Hashfunktion gut gewählt wird, können Overflow Listen vermieden werden
  → hsearch(k) benötigt 1 I/O Operation, hinsert, hdelete benötigt 2 I/O Operationen
- Eine Hashfunktion soll nicht eindeutig sein, aber sie soll die gegebenen Daten gleichmäßig verteilen. → solche Hashfunktionen sind schwer zu entdecken
- **Einfache Varianten** von Hashfunktionen:
  - $h(k) = k \; mod \; N$ wobei N eine Primzahl sein sollte.
    (Alternativ: Bei $N = 2^d$ werden die letzten d bits von k berücksichtigt)
  - $h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor$
- Wächst die Datenmenge, so werden die I/O-Kosten unvorhersehbar wegen Overflow Listen. Sinkt die Datenmenge, so wird viel Speicherplatz verschwendet.
  → regelmäßiges Rehashing erforderlich → teuer und Index ist solange blockiert

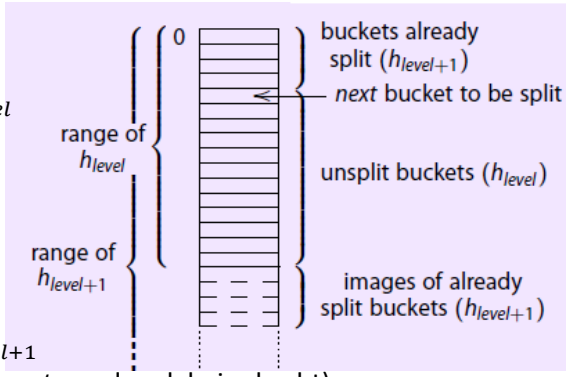
6.3 Extendible Hashing

- Kann sich an wachsende und sinkende Datenmengen flexibel anpassen
- **Global depth** n: Die letzten bits von h(k), um einen Bucketzeiger nachzuschauen im Directory
- **Local depth** d: Die Hashwerte h(k) von allen Einträgen in diesem Bucket unterscheiden sich in ihren d letzten Bits.
- Einfügen eines Records:
  - h(k) anwenden und die letzten n bits nützen, um den Bucketzeiger zu finden.
  - Wenn das Bucket noch Kapazität hat, dann hinzufügen, ansonsten Bucket teilen und Anzahl der Bits erhöhen, damit diese sich unterscheiden.
- Verwendet Overflow Listen nur im Extremfall, falls die Hashfunktion zur Aufteilung nicht ausreicht.
- Werden Einträge gelöscht, so kann ein Merge angestoßen werden, bei dem 2 Buckets wieder vereint werden.



6.4 Linear Hashing

- Ähnlich zu Extendible Hashing
- benötigt kein Hash Verzeichnis, flexible Kriterien, wenn ein Bucket aufgeteilt werden soll
- ist die Verteilung schlecht, so wird es uneffizient
- Idee: Benütze eine Familie von Hashfunktionen h_0, … Dabei ist die Range $h_{level+1}$ doppelt so groß wie von h_{level}
- Üblicher Ansatz für Hashfunktion Familie: $h_{level}(k) = h(k)\ mod\ (2^{level} \cdot N)$
- Bucket teilen:
  - ein neues Bucket allokieren an Position $2^{level} \cdot N + next$
  - Einträge neu verteilen durch rehashing über $h_{level+1}$
  - next um 1 erhöhen (alle Buckets mit $position < next$ werden dabei rehasht)
  - Falls $next > 2^{level} \cdot N - 1$: $next \leftarrow 0; level{+}{+}$
- Rehashing eines Buckets benötigt auch ein rehashing der Overflow Kette
- Suche: $h_{level}(k) = \begin{cases} < next & we\ hit\ an\ already\ split\ bucket \rightarrow rehash \\ \geq next & we\ hit\ a\ yet\ unsplit\ bucket \rightarrow bucket\ found \end{cases}$



Chapter 7 – External Sorting

7.1 Query Processing

Ein Query Prozessor verwendet einen Plan bestehend aus spezialisierten Routinen (Query Operatoren), welche Zeit und Speichereffizient umgehen.

7.2 Sorting

- Hauptproblem: Wie kann man eine so große Menge an Records sortieren, die nichtmal in den Hauptspeicher passt.
- Ansatz über eine Zwei Phasen Variante
  - Sortieren einer beliebig großen Datei mit 3 Seiten im Bufferspeicher möglich machen
  - Algorithmus neu schreiben, um Nutzen von realistischem Bufferspeicher zu ziehen

7.2.1 Two-Way Merge Sort

Pass 0 (**Input:** $N = 2^k$ unsorted pages; **Output:** 2^k sorted runs)

1. **Read** N pages, **one page at a time**
2. **Sort** records, page-wise, in main memory.
3. **Write** sorted pages to disk (each page results in a **run**).

This pass requires **one page** of buffer space.

Pass 1 (**Input:** $N = 2^k$ sorted runs; **Output:** 2^{k-1} sorted runs)

1. Open two runs r_1 and r_2 from Pass 0 for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new two-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

⋮

Pass n (**Input:** 2^{k-n+1} sorted runs; **Output:** 2^{k-n} sorted runs)

1. Open two runs r_1 and r_2 from Pass $n-1$ for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new 2^n-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

- Anzahl I/O: $2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$

7.2.2 External Merge Sort

- Optimierungen durch zwei wesentliche Knoten:
  - Anzahl der initiale Läufe reduzieren, durch nützen des gesamten Bufferspeichers B
  - Anzahl der „Passes“ reduzieren indem mehr als 2 Runs parallel verschmolzen werden
- Anzahl I/O: $2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \frac{N}{B} \right\rceil\right)$
- Zugriffspattern welcher Block, als nächstes benötigt wird ist zufällig. ➔ gleichzeitig b Seiten lesen für jeden Run. Man zahlt mit einem gesunkenen fan-in (mehr I/O Operationen) ➔ in der Praxis reicht 1 Mergepass

Pass 0 (**Input:** N unsorted pages; **Output:** $\lceil N/B \rceil$ sorted runs)

1. **Read** N pages, B **pages at a time**
2. **Sort** records in main memory.
3. **Write** sorted pages to disk (resulting in $\lceil N/B \rceil$ **runs**).

This pass uses B **pages** of buffer space.

Pass n (**Input:** $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs; **Output:** $\frac{\lceil N/B \rceil}{(B-1)^n}$ sorted runs)

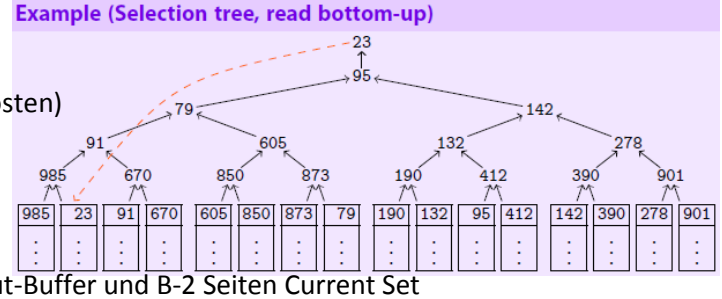
1. Open $B-1$ runs $r_1 \ldots r_{B-1}$ from Pass $n-1$ for reading.
2. **Merge** records from $r_1 \ldots r_{B-1}$, reading page-by-page.
3. **Write** new $B \cdot (B-1)^n$-page run to disk (page-by-page).

This pass requires B **pages** of buffer space.

7.2.3 Comparisons

Mittels External Merge Sort wird die CPU stärker belastet, da $B - 2$ Vergleiche notwendig sind, um das aktuell kleinste Record zu suchen.
➔ Verwendung eines Selektionsbaums (reduzierte Kosten)

Example (Selection tree, read bottom-up)



7.2.4 Replacement Sort

- Damit man initiale Runs mit mehr als B Seiten erreichen kann
- Verwendet 1 Seite Input-Buffer, 1 Seite Output-Buffer und B-2 Seiten Current Set
- Um die CPU voll auszulasten, wird doppeltes Buffering angewendet. Es gibt einen zusätzlichen **Shadow Buffer** der vorgeladen wird und sobald der Inputbuffer leer ist, kann die CPU zum Shadow Buffer wechseln und der eigentliche Inputbuffer wird wieder parallel nachgeladen.

- **Algorithmus**:
  1. Öffne einen leere Run-Datei zum Schreiben
  2. Lade nächste Seite der Datei in den Input Buffer (falls nichts mehr zum Einlesen → 4.)
  3. Solange Platz in „current set“ ist: Schiebe Records vom Input Buffer in „current set“ (falls der Input Buffer leer ist → 2)
  4. $k_{out} \leftarrow \max(\text{output buffer});\ r \leftarrow \min_{k \geq k_{out}} (\text{current set})$
     Schiebe r in den output Buffer. Bei vollem Output Buffer füge ihn zum aktuellen Run dazu.
  5. Wenn $\forall k \in$ current set: $k < k_{out}$, dann füge den Output Buffer zum aktuellen Run dazu und starte einen neuen.
  6. Falls nichts mehr zum Auslesen → STOP, sonst → 3.

7.2.5 B⁺-Trees for Sorting

- Ist der Baum clustered, dann einfach per sequential Scan ausgeben
- Ist der Baum unclustered, bringt uns dieser keine Vorteile und wir müssen sortieren, da sonst eine I/O Operation pro Record (im Extremfall) zu Buche schlagen.

Chapter 8 – Evaluation of Relational Operators

8.1 Relational Query Engines

Relational Query Engine	Virtual Machine (VM)
Operatoren der relationalen Algebra	Primitive VM Instruktionen
Operiert über Streams von Zeilen	Agiert auf Objektrepräsentationen
Operatorennetzwerk (Baum/DAG)	Sequentielles Programm (Zweige, Schleifen)
Verschiedene Varianten eines Operators	Kompakte Menge an Instruktionen

- Die spezifischen Operatorvarianten sind zugeschnitten auf spezifische Varianten der Daten (Indexe, Sortierung, Größe, verfügbarer Platz im Buffer, Buffer Ersetzungsstrategie)
- Query Optimierer kann (wie ein Compiler) entsprechende Operatoren auswählen.

8.2 Selection (σ)

Selection (σ)—no index, unsorted data

$\sigma_p(R)$

input access[1]	file scan (openScan) of R
prerequisites	none (p arbitrary, R may be a heap file)
I/O cost	$\underbrace{N_R}_{\text{input cost}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

Selection (σ)—hash index, equality predicate

$\sigma_p(R)$

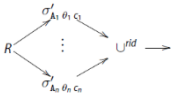
input access	hash table on R
prerequisites	r_{in} hashed on key A, p has term A = c
I/O cost	$\underbrace{sel(p) \cdot N_R}_{\text{bucket access}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

Selection (σ)—clustered B⁺-tree index

$\sigma_p(R)$

input access	access of B⁺-tree on R, then sequence set scan
prerequisites	clustered B⁺-tree on R with key k, p matches key k
I/O cost	$\underbrace{\approx 3}_{\text{B⁺-tree acc.}} + \underbrace{sel(p) \cdot N_R}_{\text{sorted scan}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

- $N_R = \left\lceil \frac{|R|}{p_r} \right\rceil$: Anzahl der Seiten in R
  $|R|$: Anzahl der Records
  p_r: Anzahl der Records pro Seite
- $0 \leq Sel(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$
- Ist der Index unclustered und sel(p) groß, dann lohnt es sich die Indexeinträge <k,rid> zu lesen und nach ihrer rid zu sortieren. Daraufhin muss jede Seite nur einmal ausgelesen werden!
- Hashindex lohnt sich nur bei A = c, insofern der Hashindex über die Spalte a gebaut wurde.
- Konjunktive Prädikate: $sel(p\ AND\ q) = sel(p) \cdot sel(q)$
- Disjunktive Prädikate: $sel(p\ OR\ q) = sel(p) + sel(q) - sel(p) \cdot sel(q)$
- Wenn alle Terme über Indexe getroffen werden, kann eine rid-basierte Vereinigung den Plan verbessern. (entsprechend bei Konjunktion Schnitt)



8.3 Projection (π)

- Im Allgemeinen wird die Größe durch Projektion einfach nur kleiner durch weggeworfene Felder, aber durch DISTINCT kann auch erzwungen werden, dass Duplikate entfernt werden.
- Duplikat Elimination:
  - Sortieren: Vorteilhaft, da Ergebnis direkt sortiert.
  - Hashbasiert: Gleiche Records sind im selben Bucket, damit nur Bucket auf doppeltes Vorkommen überprüfen. Eliminierung durch 2. Hashfunktion innerhalb des Buckets und nur die Vergleichen die den gleichen Hashwert haben.
  - Works efficiently only if duplicate elimination phase can be performed in the buffer. What to do if partition size exceeds buffer size?

8.4 Join ($\bowtie$)

- Ist eine Abkürzung für Kreuzprodukt und Selektion

8.4.1 Nested Loop Join

- Standardvorgehen von $\sigma - \times$, 3 Seiten reichen aus
- I/Os: $\boldsymbol{N_R + \underbrace{p_R \cdot N_R}_{\#tuples\ in\ R} \cdot N_S}$

Nested loops join

```
Function: nljoin (R, S, p)
/* outer relation R
foreach record r ∈ R do
    /* inner relation S
    foreach record s ∈ S do
        /* ⟨r,s⟩ denotes record concatenation
        if ⟨r,s⟩ satisfies p then
            append ⟨r,s⟩ to result
```

8.4.2 Blocked Neted Loops Join

- Random Access Kosten sparen, indem man R und S in Blocks (b_R und b_S) ausließt
- I/Os: R wird nur $\left\lceil \frac{N_R}{b_r} \right\rceil$ mal gelesen
  S wird $\left\lceil \frac{N_R}{b_r} \right\rceil$ mal gescannt mit jeweils $\left\lceil \frac{N_R}{b_R} \right\rceil \cdot \left\lceil \frac{N_R}{b_R} \right\rceil$ I/Os
- Mit Hash Table im Speicher kann der Join beschleunigt werden. Hilft nur bei equi-join!!

Block nested loops join

```
Function: block_nljoin (R, S, p)
foreach bR-sized block in R do
    foreach bS-sized block in S do
        /* performed in the buffer                    */
        find matches in current R- and S-blocks and append them
        to the result ;
```

Block nested loops join: build hash table from outer row block

```
Function: block_nljoin' (R, S, p)
foreach bR-sized block in R do
    build an in-memory hash table H for the current R-block ;
    foreach bS-sized block in S do
        foreach record s in current S-block do
            probe H and append matching ⟨r,s⟩ tuples to result ;
```

8.4.3 Index Nested Loops Join

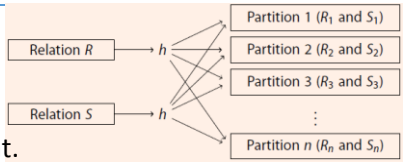
- Nimmt sich den Vorteil, indem über der inneren Relation ein Index gebildet wird. (Nur bei equi-join!!)
- Hashindex wird dabei über das Prädikat gebildet. Solche Prädikate heißen sargable (**sea**rch **arg**ument)
- Die Kosten werden abhängig von der Größe des Join-Ergebnisses. Für n treffende Tupel für ein Tupel r werden n I/Os benötigt (unclustered) bzw. $\left\lceil \frac{n}{p_s} \right\rceil$ I/Os (clustered). Zusätzlich noch N_{idx} I/Os um den ersten Treffer zu bekommen. (effektiv in der Regel: $N_{idx} \approx [1,3]$)
  ➔Index Nested Loops Join lohnt sich, alls der Join Selektiv ist!

Index nested loops join

```
Function: index_nljoin (R, S, p)
foreach record r ∈ R do
    scan S-index using (key value in) r and concatenate r with all
    matching tuples s ;
    append ⟨r,s⟩ to result ;
```

8.4.4 Sort-Merge Join

- Beide sortieren und dann mergen (siehe Mergesort). (Nur bei equi-join!!)
- Angenommen beide Relationen sind bereits sortiert und es gibt keine besonders lange Sequenzen von gleichen Schlüsseln: $\boldsymbol{N_R + N_S}$ als I/O Kosten
- Mit Blocked I/O können die I/O Operationen fast als sequentielle Reads verstanden werden
- Sortieren kann sich lohnen für Merge Join (vor allem wenn die Sortierung später noch benötigt wird)
- Im Worst-Case degeneriert Merge Join zu Nested Loop Join.



8.4.5 Hash Join

- Beide Relationen werden mit derselben Hashfunktion h gehasht. Die Entstehenden Partitionen R_i und S_i enthalten eine Teilmenge bei denen ein Match in Frage kommt. Damit kann $R_i \bowtie S_i$ mit **in-memory joins** berechnen. (Nur bei equi-join!!)
- Der innere Join verwendet meist wieder eine Hash Tabelle mit Hashfunktion h'
- R darf im Optimalfall aus maximal $(B-1)^2$ Seiten bestehen. Wegen ungleicher Verteilung der Partitionen eher kleiner, damit keine Partition mehr $B-1$ Seiten hat.
- Größere Tabellen benötigen mehrere „passes“ für die Partitionierung (rekursive Paritionierung)

8.6 Operator Pipelining

- Bisher wurden die Operatoren sequentiell ausgeführt, damit lange Antwortzeiten. ➔ mit **pipelining** werden die Ergebnisse sobald wie möglich dazu verwendet, um den nächsten Operator anzuwenden
- **Volcano Iterator Model**: Jeder Operator erhält ein Interface mit den Funktionen: open(), close(), next() (es wird immer ein neuer Tupel oder <EOF> ausgegeben)
- Manche Operatoren benötigen den gesamten Input bevor ein Ergebnis erzielt werden kann („**Blocking Operators**“ genannt). Beispiel: sum, …

A Volcano-style implementation of nested loops join $R \bowtie_p S$?

```
1 Function: open ()
2 R.open () ;
3 S.open () ;
4 r ← R.next () ;

1 Function: close ()
2 R.close () ;
3 S.close () ;

1 Function: next ()
2 while (r ≠ ⟨EOF⟩) do
3     while ((s ← S.next ()) ≠ ⟨EOF⟩) do
4         if p(r,s) then
              /* emit concatenated result   */
5             return ⟨r,s⟩ ;
      /* reset inner join input             */
6     S.close () ;
7     S.open () ;
8     r ← R.next () ;
9 return ⟨EOF⟩;
```

Chapter 9 – Cardinality Estimation

9.1 Cardinality Estimation

- Optimizer führt eine kostenbasierte Suche durch nach dem günstigsten Plan.
- Die Bestimmung der Kardinalität einer (Sub-)Query dabei äußerst wichtig.
- Es gibt 2 prinzipielle Ansätze:
  - **Datenbase Profile**: Statistische Informationen über Anzahl und Größe der Tupel, Verteilung, etc. werden für Relationen gehalten. Typisch nimmt das Model wird Unabhängigkeit und Gleichverteilung an, was in der Regel zwar falsch, aber doch eine genügend gute Abschätzung ermöglicht. (manchmal auch Histogramme)
  - **Sampling Techniques**: Query auf kleinem Beispiel ausführen und die Kosten entsprechend hochrechnen auf die gesamte Relation.

Typical database profile for relation R

$\|R\|$	**number of records** in relation R
N_R	**number of disk pages** allocated for these records
$s(R)$	average **record size** (width)
$V(A,R)$	**number of distinct values** of attribute A
$MCV(A,R)$	**most common values** of attribute A
$MCF(A,R)$	**frequency** of most common values of attribute A
⋮	*possibly many more*

9.2 Database Profiles

- Es gibt 3 mögliche Annahmen:
  - **Gleichverteilung und Unabhänigkeit**
  - Worst Case (unrelatistisch)
  - Vollständiges Wissen (unrealistisch)

9.3 Estimating Operator Cardinality

- Im Folgenden Beispiele bei denen Überall Gleichvert. oder Unabhängigkeit angenommen wird.
- **Selection**:
  sel(A = c) = 1/V(A,R) (➔ Gleichverteilung)
  $sel(A = B) = \frac{1}{max(V(A,R),V(B,R))}$ (➔ Nimmt an, dass jeder Wert des Attributes mit weniger

eindeutigen Werten einen passenden Treffer im anderen Attribut hat → Unabhänigkeit)

$$sel(A > c) = \begin{cases} \frac{High(A,R)}{High(A,R)-Low(A,R)}, Low(A,R) \leq c \leq High(A,R) \\ 0, \text{sonst} \end{cases}$$ (→ Gleichverteilung)

- **Projektion**: (löschen von Duplikaten)

$$|Q| \begin{cases} V(A,R), \text{if } L = \langle A \rangle \\ |R|, \text{if keys of } R \in L \\ |R|, \text{no dup. elim.} \\ \min\left(|R|, \prod_{A_i \in L} V(A_i, R)\right), \text{sonst} \end{cases}$$ (→ Unabhängigkeit)

- **Mengen Operationen**: Kardinalität $|Q|$ wird abgeschätzt
- **Join**: Schwierig abzuschätzen, mittels Fremdschlüssel kann für den Join $Q \equiv R \bowtie_{R.A=S.A} S$ garantieren, dass |Q| = |S| ist, da $\pi_A(S) \subseteq \pi_A(R)$ gilt. Ansonsten:
  Ansonsten für $Q \equiv R \bowtie_{R.A=S.B} S$:

$$|Q| = \begin{cases} \frac{|R| \cdot |S|}{V(A,R)}, \pi_B(S) \subseteq \pi_A(R) \\ \frac{|R| \cdot |S|}{V(B,S)}, \pi_A(R) \subseteq \pi_B(S) \end{cases}$$

9.4 Histograms

- Realistische Datenbankinstanzen sind nicht Gleichverteilt. Mit Histogrammen kann die Verteilung approximiert werden. Hierbei gibt es zwei wesentliche Varianten:
  - **Equi-Width**: Jedes Bucket hat dieselbe Breite, die Anzahl ist beliebig
    Breite eines Buckets: $w = \frac{High(A,R)-Low(A,R)+1}{B}$
    Konstruktion: Buckets b_i berechnen und einmaliger Scan
    (sollte ein Scan nicht tragbar sein, kann man auch über eine kleine Teilmenge scannen und entsprechend hochrechnen)
  - **Equi-Depth**: Jedes Bucket hat dieselbe Anzahl an Tupel, aber Breite beliebig
    Anzahl eines Buckets: $d = \frac{|R|}{B}$
    Konstruktion: d berechnen, R nach A sortieren, scannen und Buckets bilden
- Innerhalb der Buckets wird aber wieder eine Gleichverteilung angenommen!

9.5 Statistical Views

- Statistische Informationen sind nur für existierende Tabellen vorhanden, damit verliert man mit joins an Aussagekraft. (IBM nützt für Fremdschlüssel zusätzliche Informationen)

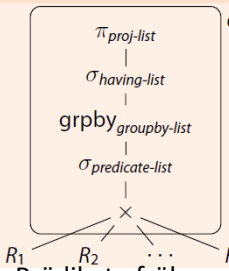
Chapter 10 – Query Optimization

- Die Aufgabe den besten Ausführungsplan zu finden ist der Heilige Gral jeder DB-Impl.
- Der Generationsprozess durchläuft 3 Schritte:
  - **Parser**: syntaktische und semantische Analyse, erstellt interne Repräsentation
  - **Rewriting**: heuristische Optimierung unabhängig von der Datenbank, Prädikate früh, vermeiden von unnötiger Eliminierung von Duplikaten
  - **Optimizer**: Optimierung Optimierungen die auf dem Kostenmodell und Informationen über den aktuellen Datenbankzustand beruhen. Vorgehen:
    - Zählt alle möglichen (aussichtsreichen) Ausführungspläne auf
    - Bestimmt deren Kosten (Qualität) und wählt den Besten aus.

Deriving a query block from a SQL SFW block

```
SELECT proj-list
  FROM R1, R2, ..., Rn
 WHERE predicate-list
 GROUP BY groupby-list
HAVING having-list
```

→



- Die Kosten eines Plans ermitteln sich auf Basis von folgenden Kostenfaktoren:
  - Die Anzahl der I/Os die benötigt werden
  - Die CPU-Kosten des Plans
  - Die Antwortzeit sowie die gesamte Ausführungszeit
- Alle der obigen Faktoren hängen von einem ab: der Größe der Zwischenergebnisse des Querys
- Ein Join über n+1 Relationen benötigt n binäre Joins. Dabei gibt es C_n mögliche Kombinationen:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{n!}$$

Possible bushy join trees joining n relations

number of relations n	C_{n-1}	join trees
2	1	2
3	2	12
4	5	120
5	14	1,680
10	4,862	17,643,225,600

10.1 Dynamic Programming

- **Idee**: Finde den günstigsten Plan für einen n-fachen Join in n Schritten. In jedem Schritt k, finde den besten Plan für alle k Subqueries. Erstelle die Pläne in Schritt k für die besten i Relationen und $k - i$ Relationen Unterpläne die im Schritt davor gefunden wurden.
- **Annahme**: Um den globalen Plan zu finden reicht es aus nur die optimalen Teilpläne zu betrachten. (Prinzip der Optimalität)
- **Vorgehen**:

Pass	4-Table Join	
1	Finde den besten Pfad zu jeder Relation R_i individuell. (index scan, full table scan, …)	
2	Finde für alle Paare R_i und R_j die beste Reihenfolge für den Join: $optPlan(\{R_i, R_j\}) \leftarrow$ best of $R_i \bowtie R_j$ and $R_j \bowtie R_i$	12 plans
3	Für jeweils 3 Relationen bestimme den besten Join, indem die optimalen Pläne aus Pass 1 & 2 verwendet werden: $optPlan(\{R_i, R_j, R_k\}) \leftarrow$ best of $R_i \bowtie optPlan(\{R_j, R_k\})$, $optPlan(\{R_j, R_k\}) \bowtie R_i, R_j \bowtie optPlan(\{R_i, R_k\}), \ldots$	24 plans
4	Für jeweils 4 Relationen bestimme den besten Plan mittels Pass 1, 2, 3	14 plans

- Mit jedem Schritt werden Kandidaten ausgeschlossen und mittels Heuristiken können Fälle wie kartesische Produkte ausgeschlossen oder nur „**left-deep**“ Pläne betrachtet werden
- „left-deep“ Pläne verwenden viele DBMS, da die innere Relation damit immer eine Basisrel. und damit **Index Nested Loops Join** möglich sind. Ermöglicht auch einfaches **pipelining**.
- *Komplexität*: Zeit: $O(3^n)$, Speicher: $O(2^n)$
  ➔ für viele Relationen TEUER!!
  ➔ Greedy Join Enumeration übernimmt hier

Find optimal n-way bushy join tree via dynamic programming

Function: `find_join_tree_dp` $(q(R_1, \ldots, R_n))$
for $i = 1$ **to** n **do**
  $optPlan(\{R_i\}) \leftarrow$ `access_plans` (R_i) ;
  `prune_plans` $(optPlan(\{R_i\}))$;
for $i = 2$ **to** n **do**
  foreach $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do**
    $optPlan(S) \leftarrow \varnothing$;
    foreach $O \subset S$ with $O \neq \varnothing$ **do**
      $optPlan(S) \leftarrow optPlan(S) \cup$ `possible_joins` $\left[optPlan(O) \bowtie optPlan(S \setminus O) \right]$;
    `prune_plans` $(optPlan(S))$;

Greedy join enumeration for n-way join

Function: `find_join_tree_greedy` $(q(R_1, \ldots, R_n))$
$worklist \leftarrow \varnothing$;
for $i = 1$ **to** n **do**
  $worklist \leftarrow worklist \cup$ `best_access_plan` (R_i) ;
for $i = n$ **downto** 2 **do**
  // $worklist = \{P_1, \ldots, P_i\}$
  find $P_j, P_k \in worklist$ and $\bowtie_\ldots$ such that $cost(P_j \bowtie_\ldots P_k)$ is minimal ;
  $worklist \leftarrow worklist \setminus \{P_j, P_k\} \cup \{(P_j \bowtie_\ldots P_k)\}$;
// $worklist = \{P_1\}$
return single plan left in $worklist$;

10.2 Greedy Join Enumeration

- Suche in jeder Iteration den günstigsten Join aus der möglich ist und arbeite damit weiter.
  (→ ähnlich zu Huffman Codierung)
- Komplexität: Zeit: $O(n^3)$, da n Iterationen und es werden alle vorhanden Paare betrachtet $O(n^2)$
- **Andere Techniken**:
  Randomisierte Algorithmen („Hill Climbing“, …)
  Generische Algorithmen

10.3 Physical Plan Properties

- Bisherige Techniken betrachten nicht die physischen Eigenschaften. Ein Indexscan kann sich lohnen, wenn dieser für einen darauf folgenden Join verwendet wird. (vgl. Sortierung)
- Optimizer halten solche Eigenschaften immer bei ihren Kandidaten notiert, um diese zu berücksichtigen.

10.4 Query Rewriting

- Indem man den Querygraphen umschreibt, kann die effizienz der nachfolgenden Optimierung deutlich erhöht werden. Dazu gehören:
  - Prädikate nach unten schieben
  - Prädikate umformulieren (z.B. $a * 100 < 5$ → a < 0.05, $a = b$ AND $b = c$ →… AND $a = c$
  - Querys entschachteln
- Verschachtelte Querys können teuer ausfallen, falls diese voneinander abhängig sind. Mittels explizitem Join lassen sich diese erst richtig optimieren.

Chapter 11 – Transaction Management

11.1 ACID Properties

Atomicity	Entweder alle oder keine Updates einer Transaktion werden ausgeführt.
Consistency	Jede Transaktion bringt die DB von einem konsistenten Zustand in einen anderen.
Isolation	Eine Transaktion darf keine Effekte von anderen Transaktionen sehen.
Durability	Eine erfolgreich abgeschlossene Transaktion bleibt dauerhaft.

11.2 Anomalies

- **Lost Update**: Der Effekt einer Transaktion geht verloren durch überschreiben.
- **Inconsistent Read**: Lesen eines temporär inkonsistenten Datenbankzustands.
- **Dirty Read**: Lesen eines Zustands, der durch Abbruch ungültig wird.

11.3 The Scheduler

- Der Scheduler entscheidet über die Reihenfolge in der auf die Datenbank zugegriffen wird.
- Eine DB Transaktion T ist eine (strikt geordnete) Sequenz $\langle s_1, \dots, s_n \rangle$ von Schritten. Jeder Schritt s_i ist ein Paar aus Zugriffsoperation a $(read/write)$ und Objekt e (Tabelle).
- Ein Schedule S gibt für eine Menge an Transaktionen T_i die Ausführungsreihenfolge aus.

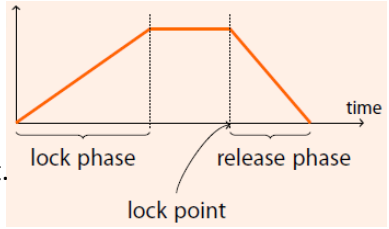
11.4 Serializability

- **Serielle Ausführung** bezeichnet die Ausführung von Transaktionen hintereinander ohne Überlappung. → kein gleichzeitiger Zugriff → **Jede Serielle Ausführung ist korrekt.**
- Eine Gleichzeitige Ausführung ist korrekt, falls bei serieller Ausführung dasselbe Ergebnis erreicht wird.
- Zwei Operationen haben einen **Konflikt**, wenn ihre Ausführungsreihenfolge eine Rolle spielt. Bzw. Formal: 2 Operationen (T_i, a, e) und (T_j, a', e') haben einen **Konflikt** in S, wenn
  - sie zu verschiedenen Transaktionen gehören ($T_i \neq T_j$), und
  - sie auf dasselbe Datenbankobjekt ($e = e'$) zugreifen und
  - mindestens eine von ihnen eine Schreiboperation ist.
    → (T_i, a, e) $\prec_S$ (T_j, a', e')
- Die Reihenfolge der Operationen innerhalb einer Transaktion darf nicht verändert werden.
- Ein Schedule S ist **Konfliktserialisierbar**, wenn der Konflikt äquivalent zu einem seriellen Schedule S' ist.

- Mittels **Konfliktgraphen** kann man effektiv testen ob ein Schedule korrekt ist. Dieser darf keine Zyklen vorweißen, sonst ist er nicht Konfliktserialisierbar. Jede Transaktion erhält einen Knoten. Gibt es $(T_i, a, e) \prec (T_j, a', e')$, so gibt es eine Kante $T_i \rightarrow T_j$.

11.6Query Scheduling & Locking

- **Idee**: Jede Transaktion benötigt einen Lock bevor auf das Objekt zugegriffen werden kann. Ist ein Objekt bereits gelockt, dann wird die aktuelle Transaktion solange blockiert. Der Scheduler schiebt die Ausführung der blockierten Transaktion auf bis der Lock freigegeben wird. Dann wird die Transaktion weiter ausgeführt.
- Reicht aber noch nicht aus.



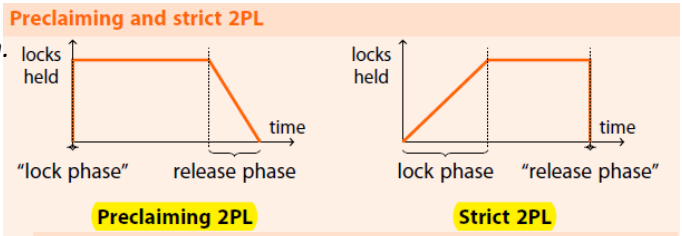
11.7 Two-Phase Locking

- Das **Two-Phase Locking** hat eine zusätzliche Bedingung wie die Abb. zeigt.
- **Theorem**: Das Nutzen von 2PL führt immer zu korrekten und serialisierbaren Schedules oder zu einem Deadlock!
- Systeme verwenden in der Regel eine Unterscheidung zwischen „**read-**“ (shared) und „**write-locks**“ (exclusive). Es ist eine sichere Operation in 2PL einen „shared“ lock in einen „exclusive“ Lock umzuwandeln.
- **Multi-Granularity Locking**: Lockt man dabei genauer (z.B. Zeilen, Seiten, Tabellenweiße), so können mehrere User auf dieselben Tabellen zugreifen, gleichzeitig steigt aber der Overhead diese Locks überhaupt zu speichern oder zu verwalten.

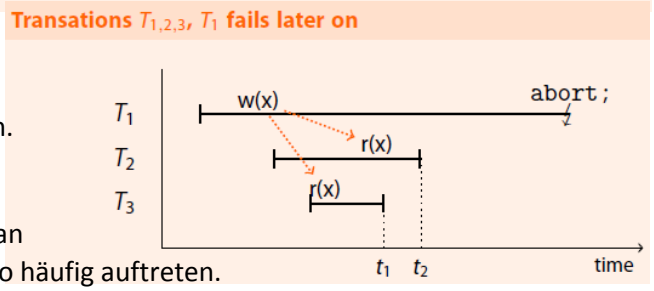
	S	X	IS	IX
S		×		×
X	×	×	×	×
IS		×		
IX	×	×		

- **Intention Lock**: Die Effizienz der Multigranularität wird damit erhöht indem es gibt für den jeweiligen Level zusätzlich IS und IX gibt, die angeben ob Locks auf dem niedrigeren Level vorhanden sind.
- 2PL schützt nicht vor **Deadlock**s. Deadlocks werden wie folgt behandelt:
  1. Überprüft periodisch das System auf Zyklen im Konfliktgraph
  2. Wahl einer oder mehr Transaktionen die abgebrochen werden müssen
     - Entweder eine Junge Transaktion abbrechen, was dazu führen kann dass diese wieder und wieder abgebrochen wird oder
     - eine alte Transaktion, was unter Umständen viel Berechnungszeit wegwirft und zusätzlich das Rückgängigmachen noch viel kostet

Alternativ: Über Timeout Transaktionen abbrechen.



- Ordnet man alle Datenbankobjekte, so können Deadlocks vermieden werden. Ansonsten hilft es auch das „preclaiming“ 2PL Protokoll Strategie zu verwenden.
- **Cascading Rollback**: →
  Mit Strict 2PL können cascading Rollback vermieden warden, da sichergestellt wird, dass geschriebene Daten nicht vor dem commit gelesen werden können.



11.8 Optimistic Concurrency Protocol

- Bisheriger Ansatz äußerst pessimistisch. Hier geht man davon aus, dass nicht serialisierbare Konfilkte nicht so häufig auftreten.
- Spart den Overhead der durch Locks erzeugt wird und investiert nur, wenn es wirklich benötigt wird. Erst beim Commit wird untersucht ob ein Konflikt vorlag.

- Vorgehen:
  - **Read Phase**: Ausführen einer Transaktion, aber kein direktes Zurückschreiben auf den Datenträger, sondern sammeln der Updates in privatem Workspace.
  - **Validation Phase**: Beim Commit einer Transaktion wird überprüft ob nur akzeptable Konflikte augetreten sind. Ansonsten wird die Transaktion abgebrochen.
  - **Write Phase**: Übertragen der Daten vom privaten Workspace in die DB.
- Die Validation und Write Phase benötigt eine ununterbrochene kritische Session.

11.9Multi-Version Concurrency Control (MVCC)

- Timestampmechanismen: Jeder Transaktion erhält einen Timestamp (system clock oder DBMS-internal counter)
- Timestampmechanismen werden dazu verwendet, um eine Ordnung der Transaktionen herzustellen und zu entscheiden welche Zeilen sichtbar sind.
  → **Multi-Version Conucrrency Control** (MVCC)
- Benötigt Garbage Collection und mehr Speicher ist aber dafür frei von Locks.
- Verschiedene Transaktionen lesen verschiedene Objektversionen.
- MVCC verwendet „**Snapshots**“, um festzustellen, welche Version für jedes Datenbankobjekt für die aktuelle Transaktion sichtbar ist. Snapshot wird erstellt, wenn die Transaktion startet. Um einen Snapshot zu erstellen, wird die *höchste XID aller übertragenen Trans*aktionen sowie die *Liste aller XIDs aller Transaktionen* die aktuell ausgeführt werden benötigt.
- Um zu entscheiden welche Zeilenversion aktuell ist, wird jedem Objekt (in PostgreSQL einer Zeile) eine xmin (Erstellung der Zeile) und xmax (Löschung der Zeile) zugeordnet. (Updates werden über löschen und neuerstellen simuliert) Wichtig: Keine physikalische Löschung!!!
- Garbage Collection wird verzögert, bis das entsprechende Objekt für alle Transaktionen unsichtbar wird.