Datenbankdesign

Andreas Hofer

March 18, 2025

Inhaltsverzeichnis:

1 Kanonische Überdeckung

Eine kanonische Überdeckung F_c ist eine minimale Menge von funktionalen Abhängigkeiten welche noch äquivalent zu der Menge F sind. Da im schlechtesten Fall die kanonische Menge die Menge selbst ist, hat jede Menge von funktionalen Abhängigkeiten eine kanonische Menge. Eine Menge kann auch mehrere kanonische Überdeckungen haben.
Eine Menge ist eine kanonische Überdeckung wenn:

1. $F_c^+ = F^+$ -> Die beiden Mengen sind äquivalent
2. F_c hat keine weiteren Abhängigkeiten $X \rightarrow Y$ welche überflüssig sind.
3. Alle linken Terme sidn einzigartig

1.1 Vorgang

Man berechnet eine kanonische Überdeckung indem man folgende Schritte ausführt:

1. Linksreduktion
   - Die Anzahl der linken Terme wird reduziert
2. Rechtsreduktion
   - Die Anzahl der rechten Terme wird reduziert
3. Entfernen von leeren Mengen
4. Vereinigung
   - Man wendet eventuell anfallende Armstrong-Axiome selbst an.

Zu beachten ist, dass in diesem Prozess eine der möglichen kanonischen Überdeckungen gefunden wird.

1.1.1 Linksreduktion

Für jedes $X \rightarrow Y$ wird eine Linksreduktion durchgeführt:

- Für alle Attribute A von X wird überprüft ob:
  - $Y \subseteq H(F, X - A)$
  - Ist Y äquivalent zur Hülle $X - A$
- Wenn das gilt, dann wird X durch $X - A$ ersetzt -> $X - A \rightarrow Y$

Ein paar Regeln zu Vereinfachung des Vorgangs ist, dass, wenn links jeweils nur ein Attribut steht, diese nie reduziert werden können. Wenn man

1.1.2 Rechtsreduktion

Für jedes $X \rightarrow Y$ wird eine Rechtsreduktion durchgeführt:

- Für alle Attribute B wird überprüft ob:
  - $B \in H(F - \{X \rightarrow Y\} \cup \{\})$

1.1.3 Entfernung der leeren Mengen

Alle FDs der Form $X \rightarrow$ werden entfernt, welche eventuell durch die Rechtsreduktion entstanden sind.

1.1.4 Vereinigung

Neue FDs, welche entstanden sind, sollten vereinfacht werden.
`X->A, X->C, X->F wird zu X->ACF`

1.1.4.1 Beispiel

2 Datenbankzerlegung (Decomposition)

Da eine Datenbank sich (wenn sie aktiv verwendet wird) stets verändert, muss man diese regelmäßig verändern um weiterhin einer Normalform zu entsprechen. Bei der Zerlegung kann es speziell bei höheren Normalformen passieren, dass man eine Abhängigkeit nicht weiter abbilden kann. Also kann es unter Umständen passieren, dass eine funktionale Abhängigkeit aufgegeben werden muss. Was wiederum **niemals** passieren darf, ist, dass Daten verloren gehen. Alle Daten weiter zu besitzen ist das wichtigste Gut bei der Überführung einer Datenbank.
Bei der Zerlegung wird eine Tabelle in seine Schemata übersetzt. Diese müssen am Ende vereint wiederum die Tabelle ergeben.
Eine Zerlegung hat zwei Gütekriterien, die Verlustlosigkeit und die Abhängigkeitsbewahrung.

2.1 Verlustlosigkeit

Eine Zerlegung ist genau dann verlustlos, wenn ein Join aller Teile wiederum zur Tabelle führt: $\pi_{sch(R_1)}(R) \bowtie \pi_{sch(R_2)}(R) \bowtie ... \bowtie \pi_{sch(R_n)}(R) = R$
Da man hier einen natural join verwendet, wodurch es seinen Partner findet, sollte es automatisch wieder zur Tabelle vereint werden. Bei der Verlustlosigkeit muss beachtet werden, dass es stets nur um die Bewahrung der Information und nicht der Tupel geht. Wenn eine Tabelle zerlegt wird, werden Duplikate entfernt.
Bei der Bestimmung der Verlustlosigkeit sollte man zuerst die gegebene Instanz untersuchen, die Tabelle aufteilen und danach überprüfen ob diese verlustfrei war. Wenn das gegeben ist kann man es allgemein überprüfen ob es für alle Instanzen gilt.
Dabei wird mit der Hülle überprüft, ob

2.1.1 Enschränkungen

Einschränkungen bestimmen, für welche Relationen Das geschieht durch einen bestimmten Prozess, wobei für jede echte Teilemenge der gleiche Prozess durchlaufen wird:

1. Triviale FDs werden entfernt (Wie zB. $X \to X$)
2. Nur die Attribute beachten, welche vorkommen
3. Falls Attribute vorhanden sind, werden diese hinzugefügt.

2.1.1.1 Beispiel
Gegeben ist das Schema F_R=$\{A \to D, D \to C, C \to D\}$ mit der Zerlegung von R zu R_1[A, C] und R_2[C, D].
Zur Berechnung der Einschränkung sowie der Abhängigkeitsbewahrung muss zuerst die Hülle der Teile der Zerlegung berechnet werden:
H(F, A)
{A}={D, C} `Geschnitten mit A, C ergibt das A -> C`
H(F, C)
{C}={D} `Geschnitten mit A, C ergibt C->0 und mit C, D ergibt C->D`
H(F, D)

{D}={C} `Geschnitten mit C, D ergibt D->C`
Aus diesem Grund sind die Einschränkungen von R_1 $A \to C$ und von R_2 $C \to D, D \to C$
Wenn man nun herausfinden will, ob die Abhängigkeiten bewahrt werden, muss man überprüfen ob $F_R^+ = (F_{R_1} \cup F_{R_2})$ also ob $\{A \to D, D \to C, C \to D\}^+ = \{A \to C, C \to D, D \to C\}^+$
Das erste Kriterium kann als gegeben angenommen werden, da sowohl F_{R_1} als auch F_{R_2} aus F_R hergeleitet wurde. (Vorrausgesetzt man hat richtig gerechnet.)

3 Normalformen

Diese Zerlegungen sowie die Überprüfung ob Information verloren gegangen ist dient dazu eine Datenbank in eine bessere Form zu bringen. Eine dieser Güteklassen sind die Normalformen. Dabei existieren die erste bis zur fünften Normalformen (1NF, 2NF, 3NF, 4NF, 5NF) sowie die sogenannte Boyce-Codd Normalform (BCNF). In der Praxis finden jedoch meist nur die ersten drei, sowie die Boyce-Codd Normalform Anwendung. Für die vierte und fünfte ist der Rechenaufwand in Relation zu den Vorteilen unverhältnismäßig hoch. Aus diesem Grund werden diese auch in dieser Vorlesung nicht behandelt. Es sollte beachtet werden, dass Normalformen stets niedrigere Normalformen auch enthält. Also hat die zweite Normalform ebenfalls die Güte der ersten Normalform.

3.1 Normalisierung

Die Normalisierung ist der Prozess die Normalform einer Datenbank zu erhöhen. Dabei werden meist Attribute eines (schlechten) Schemas in kleinere (gute) Schemas aufgeteilt werden, welche den Normalformen genügen. Die Qualität einer Datenbank kann jedoch nur aufgrund verschiedener Tests bestimmt werden und besagt nicht, dass es auch ein gutes Schema hat.
Eine Denormalisierung passiert hingegen, wenn mehrere Relationen einer Normalform als Relation gespeichert werden. Dieses Ergebnis hat höchstwahrscheinlich eine niedrigere Normalform als dessen Ursprung.

3.2 Erste Normalform / 1NF

Die erste Normalform ist die niederwertigste Normalform und verbietet zusammengesetzte oder mehrwertige Attribute. Strategien um die 1NF zu erreichen können sein:

- Aus zusammengesetzten Attributen: Jeden Teil zum eigenen Attribut konvertieren
- Mehrwertige Attribute: Neues Tupel für jeden Wert des mehrwertigen Attributs
- Geschachtelte Relationen: Erzeuge ein neues Tupel für jedes Tupel der geschachtelten Relation

MName	MNum	MSVN	Standort
Martin	1	12345678	{Graz, Linz, Salzburg}

3.3 Zweite Normalform / 2NF

Die zweite Normalform wird erreicht, wenn jedes nicht-Schlüsselattribut voll funktional abhängig von allen Kandidatenschlüsseln ist. In anderen Worten darf es keine Spalte geben, welcher nicht durch den Kandidatenschlüssel eindeutig identifizierbar ist. Die mathematische Definition ist: $K \to A \iff K \to A \wedge \forall X \subset K : X \not\to A$ Strategien sind erneut:

- Für jeden Teilschlüssel eine neue Relation mit seinen abhängigen Attributen generieren.

Während jedoch keine direkten Abhängigkeiten erlaubt sind, können diese immer noch transitiv voneinander abhängig sein.

3.4 Dritte Normalform / 3NF

Die dritte Normalform wird erreicht, wenn jeder Superschlüssel X mindestens eine der Vorraussetzungen erfüllt:

1. X ist trivial
2. X ist ein Superschlüssel von Y
3. Es darf kein Attribut geben, welches in der Relation nicht vom Kandidatenschlüssel X abhängig ist.

3.4.1 Zerlegung in 3NF

Man kann mittels des Synthesealgorithmus eine große Tabelle in kleinere Tabellen zerlegen. Diese Tabellen sind danach alle in der dritten Normalform. Gleichzeitig ist diese Zerlegung auch Verlustfrei und abhängigkeitsbewahrend.
Zur Berechnung muss man:

1. Die Kanonische Überdeckung berechnen
2. Für jede Funktionale Abhängigkeit aus der kanonischen Überdeckung:
   - Erstelle eine Relation welche die Abhängigkeit $X \to Y$ vereinigt.
   - Diese Relation erhält die FDs
3. Falls keine der neuen Relationen einen Kandidatenschlüssel enthält, wähle einen K aus dem Schema.
   - Erstelle dadurch eine Relation mit dem Schema K
   -

3.4.2 Boyce-Codd-Normalform / BCNF

Die Boyce-Codd-Normalform (BCNF) und die 3NF haben relativ ähnliche Vorraussetzungen, jedoch schränkt die BCNF das Schema noch etwas mehr ein. Während bei der dritten Normalform jeder Kandidatenschlüssel eine der drei Vorraussetzungen erfüllen musste, ist die dritte in der BCNF nicht mehr zulässig. Dadurch kann in der BCNF nicht mehr alle funktionalen Abhängigkeiten bewahrt werden, was bis zur 3NF möglich war.

3.4.2.1 Dekompositionsalgorithmus

Der Dekompoisitionsalgorithmus dient zur Konvertierung einer Relation in die BCNF. Hierbei werden jedoch eventuell nicht alle Funktionalen Abhängigkeiten bewahrt. Die Schritte sind:

1.

3.5 Übung:

R[A, B, C, D, E, G] mit F = $\{A \to BD, AB \to E, B \to EG, C \to AB\}$
Linksreduktion: Da nur AB mehr als ein Element besitzt, kann man dieses als einziges eventuell reduzieren. Wenn man das A bei $AB \to E$ wegstreichen will, muss man zuerst die Hülle berechnen um das zu bestätigen. Dazu muss man die Hülle bestimmen um nachzuprüfen, ob man trotzdem auf die gleiche Abhängigkeit kommt. $\not{A}B \to E$ -> $H(F, B)$={E, G} -> `Deshalb kann man A weglassen.` $A\not{B} \to E$ -> $H(F, B)$={B. D, E, G} -> `Deshalb kann man auch B weglassen. Aus diesem Grund kann man die gesamte Abhängigkeit weglassen` Rechtsreduktion: Hierbei kann man nicht sofort das Ergebnis sehen, aber ungefähr erkennen welche Teile reduziert werden können. $C \to AB$ kann eventuell reduziert werden. Hierbei muss man wieder die Hülle berechnen, wobei jedoch alle Attribute der Relation gegeben sein müssen.
$C \to A\not{B}$ -> H(F, C)={A, B, D, E, G} `B kann reduziert werden.`
Jetzt hat man die kanonische Überdeckung $F_C = \{A \to BD, B \to EG, C \to A\}$ und aus dieser kann man die neuen Relationen generieren. $R_A[A, B, D], F_A = \{A \to BD\}$
$R_B[B, E, G], F_B = \{B \to EG\}$
$R_C[C, A], F_C = \{C \to A\}$

4 Normalform - Cheat Sheet

Normalform	Vorraussetzung
1NF	Keine Zusammengesetzten/Mehrwertigen Attribute
2NF	Alle Attribute durch Kandidatenschlüssel identifizierbar

5 Speichermedien

Da sich Datenbanken stets mit der Speicherung von Daten befassen, ist die Art des Speichermediums stets eine relevante Überlegung. Dabei unterscheidet man zwischen verschiedenen Faktoren, unter anderem Kosten, Kapazität und Vefügbarkeit.
Zusätzlich unterscheidet man zwischen den unterschiedlichen herarchischen Ordnungen von Speicher (Siehe Betriebssysteme Grundlagen). Dabei bildet der Speicher eine Pyramide welche einen Kompromiss aus Schreibgeschwindigkeit und Kapazität darstellt.
Jedes Speichermedium benötigt außerdem noch einen Controller, welcher die Schnittstelle zwischen der physischen Platte und dem Computer darstellt (Siehe Betriebssysteme Grundlagen). Auf Festplatten Diese sind:

- Cache
  - Schnellster aber auch teuerster Speicher. Wird direkt von der CPU verwaltet
- RAM (Hauptspeicher)
  - Trotzdem noch schneller Zugriff (< 100ns) mit Datentransferraten von 25 GB/s. Dieser Speicher ist jedoch meist zu klein um eine gesamte Datenbank zu speichern.
- Festplatte
  - Der erste nicht-flüchtige Speicher (Verfällt nicht sobald kein Strom gegeben ist.)
  - Bedeutend langsamer als RAM oder Cache (Im ms Bereich)
  - Daten werden auf Magnetscheiben gespeichert.
  - Kostet nur mehr 25€/TB (Statt etwa 4000€ für RAM)
  - Im Schnitt verdoppelt sich die verfügbare Kapazität alle 2 bis 3 Jahre
- Solid State Drive (SSD)
  - Bedeutend schneller mit geringeren Zugriffszeiten als die HDD
  - Schnellester persistenter Speicher
  - Im Embedded Systems schon seit längerem verbreitet.
  - Spezialform: EEPROM ((Electronically Erasable Programmable Read-Only Memory)) kann durch elektrische Impulse neu beschrieben werden.
  - Hat verschiedene Zugriffsmuster:
    - Random-access: Kann in beliebiger Reihenfolge gelesen werden.
    - Block-based: Nicht in Bytes sondern in Blöcken (Von z.B. 4096 Bytes)
  - Optische Medien
    - Daten werden von einer Platte via Laser gelesen und geschrieben.
    - Hat verschiedene Typen. Von CDs (640MB) bis Blu-Ray (25GB) mit unterschiedlich schnellen Zugriffszeiten

- Magnetband
  - Rein sequentieller Speicher. Dadurch sehr langsam.
  - Kostet nur mehr 10€/TB

Die Forschung befasst sich momentan damit die Zugriffszeiten von persistenten Speichern zu reduzieren. Ein solches Produkt ist NVRAM (Non-Volatile Memory), welcher eine bedeutend geringere Latenz besitzt. Er ist zwar bedeutend günstiger als Ram, ist mit maximal 512GB an Größe jedoch bedeutend kleiner. Solche NVRAM-Speicher sind für Datenbanken sehr interessat, da sie viel schnellere Zugriffszeiten ermöglichen.

5.1 Festplatten

Aufgrund ihrer geringen Kosten und relativ schnellen Lesegeschwindigkeiten sind Festplatten das wichtigste Medium für Datenbanken.

5.1.1 Zugriffszeiten

Zur Berechnung der Zugriffszeit eines Speichers muss man einige Schritte beachten:

- Seek Time
  - Die Zeit um den Lesekopf auf die richtige Spur zu bewegen
  - Rotational Latency
    - Die Zeit die benötigt wird damit der relevante Sektor den Lesekopf erreicht.
  - Lesezeit
    - Die Zeit die benötigt wird um die Datei auszulesen

Die Latenz wird als Summe dieser drei Faktoren berechnet.
Die Seek Time wird aus der Drehzahl berechnet, wobei man den worst und average case beachtet. Abhängig von der Drehzahl kann man die maximale Zeit mit $L = 60/Drehzahl$ berechnen.
Ein weiteres Qualitätskriterium ist die Mean Time Before Failure (MTBF) was die statistische Wahrscheinlichkeit, dass die Festplatte ausfällt, darstellt.

5.1.2 Sektoren

Zusammenhängende Reihen auf Festplatten werden Blöcke genannt. Abhängig von der Größe der Festplatte kann es zu unbeschreibbaren Lücken zwischen den Blöcken kommen (Interblock Gaps). Diese Blöcke werden in der Regel mit einer Größe von 4096 Byte generiert. Dies hat einen historischen Grund, da 4KB eine Größe war, welche von jedem Betriebssystem unterstützt wurde. Da Programme mit der Zeit immer größer werden, wird es auch relevanter die Größe der Speicherblöcke zu erhöhen.

5.2 Solid State Drive (SSD)

Daten auf der SSD werden auf Seiten abgelegt, welche ebenfalls in Blöcke unterteilt sind. Dadurch spiegelt es HDDs wieder und man merkt so keinen Unterschied ob man eine HDD oder SSD verwendet (Außer bei Schreibgeschwindigkeiten). Man muss beachten, dass SSDs keinen Block neu beschreiben können sondern stets einen Block löschen muss um ihn danach komplett neu zu beschreiben. Da die Lebensdauer einer SSD anhand ihrer Löschvorgänge beschränkt ist, werden zu löschende Daten in Erase Blocks gebündelt um dann zusammen gelöscht zu werden. Wenn man ein File also 'Updated' wird in der SSD der Speicherblock an einen anderen Ort verschoben und der alte Block zum Löschen vorgemerkt.

5.2.1 Wear Levelling

Eine Methode zur Verlängerung der Lebensdauer einer SSD ist das Wear Levelling. Dabei wird gespeichert wie oft ein Block bereits gelöscht worden ist und anhand dieser Information werden Erase Blocks erstellt. So kann man sicherstellen, dass alle Speicherblöcke gleichermaßen benutzt werden um so nicht einen Teil der SSD schnell zu verlieren.

5.2.2 Praxis

Die Performance einer SSD hängt von vielen internen Effekten ab. Eine volle SSD ist bedeutend langsamer als eine leere. Hohe Temperaturen können zu geringeren Geschwindigkeiten führen. SSDs sind auch bedeutend schneller bei sequenziellem als bei zufälligem Zugriff. Im schlimmsten Fall kann eine SSD langsamer sein als eine HDD.

5.3 Zugriffsoptimierung

Transfer zwischen persistentem Speicher und RAM sollte möglichst effizient sein. Dafür ist das Datenbank Management System (DBMS) verantwortlich. Dieses kann so versuchen die Anzahl der Zugriffe zu minimieren sowie so wenige Blöcke wie möglich zu schreiben/lesen. Das DBMS will auch so viel Speicher wie möglich innerhalb des RAMS behalten.
Man kann jedoch auch innerhalb des Zugriffs in der Festplatte optimieren um so einen besseren Block Speicherzugriff zu erzielen. Mittels Disk Arm Scheduling kann man sicherstellen, dass bei einem Zugriff der durchgeführte Weg des Lesearms so gering wie möglich ist. Das passiert mittels Elevator-Algorithmus, welcher bei einem Pfad die geringste Distanz findet. Zusätzlich kann man bereits bei der Speicherung der Daten diesen so abspeichern, dass er beim Lesen effizient ausgelesen werden kann. Mit der Zeit geschieht jedoch stets eine Fragmentierung des Speichers, sodass die Daten erneut optimiert werden müssen.
Alternative Wege um den Durchsatz zu erhöhen sind asynchrone Optimierungsvorgänge. So kann der Controller auf die Platte schreiben, wenn gerade Ressourcen zur Verfügung stehen um diese später schneller wieder finden zu können.

5.4 Logging

Das Logging schreibt ständig den Status sowei Vorgänge der Datenbank auf einen Speicher. Hierfür gibt es spezieller Logdisks, welche sequentiell beschrieben werden um so den Speicher zu optimieren.

5.5 Puffer

Der Puffer sitzt zwischen der Festplatte und dem Hauptspeicher und kann für schnellerem Zugriff oft benützter Dateien verwendet werden:

- Anforderung an Puffer wird verwertet
- Falls der Block im Puffer nicht verfügbar ist: Speicher wird für Block reserviert.
- Block wird aus Festplatte gelesen und in Puffer zwischengespeichert und überschrieben sobald er sich verändert.

Es gibt verschiedene Strategien um Blöcke zu löschen, falls kein Speicher mehr vorhanden ist:

- Least Recently Used (LRE)
  - Der am längsten nicht verwendete Block wird ersetzt. Durch diese können Zugriffsmuster zur Abschätzung der Zukunft verwendet werden.
- Most Recently Used (MRU)
  - Der als letztes verwendete Block wird verworfen.
  - Diese Strategie kann fatal für Datenbanken sein, da so kein Puffer für oft verwendete Dateien erstellt wird.
- Pinned Block

  - Gibt einen Block an, welcher nicht gelöscht werden darf

- Toss Immediately
  - Jegliche Daten werden direkt nach Verwendung verworfen.

5.6 Dateiorganisation

Auch die Organisation der Daten selbst kann ein Faktor für die Zugriffszeiten sein. Daten sind in ihrer grundlegendsten Form nur eine Sequenz an Blöcken, wodurch man ihre Abspeicherung anhand der spezifischen Anforderungne optimieren kann.

5.6.1 Cooked vs Raw

In einem DBMS kann man entweder den Controller des Betriebssystems verwenden um Daten abzuspeichern oder selbst einen definieren. Dies nennt man entweder cooked oder raw. Cooked ist zwar einfach zu implementieren, kann jedoch womöglich unomptimiert sein. Raw lässt zwar bedeutend mehr Optimierung zu, die gesamte Abspeicherung obliegt jedoch der Datenbank wodurch es viel Aufwand ist.

5.6.2 Variable vs Fixed Length

Die Anordnung einer Datei kann entweder fixiert geschehen (Wenn x byte an Daten sequenziell existieren) oder eine variable Länge besitzen. Wenn die Datensatzlänge fixiert ist, muss man, falls eine Datei in der Mitte gelöscht wird, damit umgehen, wofür es drei Strategien gibt:

- Alle Datensätze werden um eine Position nach oben verschoben
- Der letzte Datensatz wird in die Lücke kopiert
- Jeder Datensatz hat eine Referenz auf den nächsten Datensatz (Free List)

Heutzutage wird größtenteils die Free List Strategie verwendet da man so den geringsten Performanceverlust hat.
Speziell bei varchars, welche eine variable Größe haben können, ist eine flexible Datenlänge jedoch von Nutzen. [TODO Slide 40 - 48]

6 Logische Anordnung

Die logische Anordnung von Datensätzen in der Datenbank kann auch ein großer Faktor für dessen Zugriffszeit sein. Dabei gibt es in der Regel drei Ansätze zur Datenspeicherung: Sequentiell, Heap oder Hash.

6.1 Sequentiell

Die Datensätze sind in der sequentiellen Anordung nach ihrem Suchschlüssel geordnet. Der Suchschlüssel ist in der Regel der Primary Key. Diese Datensätze sind jeweils mit Pointern verkettet. Idealerweise sind diese Datensätze nicht nur logisch (nach den Pointern) sondern auch physisch (im Speicher selbst) sortiert gespeichert. Das hat den Grund, dass es schneller ist Datensätze auszulesen wenn diese hintereinander liegen. Solch eine sequentielle Anordnung ist jedoch nach einer Zeit nicht mehr zu 100% durchführbar. Wenn man zum Beispiel ein Element löscht muss man viele Elemente neu anordnen um die gleiche Sortierung zu erhalten. Ein noch größeres Problem ergibt sich, wenn man neue Daten zwischen den alten einfügen will, der Block jedoch gefüllt ist. Dann muss die neue Datei in einen Overflow-Block verschoben werden, wodurch die physische Sortierung nicht mehr gegeben ist. Alternativ kann die Datenbank in ihrer Gesamtheit in einen neuen Block verschoben werden, das kann jedoch extrem aufwändig sein.
Eine sequentieller Speicherung kann so aussehen. Dabei ist zu beachten, dass der Verweis stets auf das nächste Element

und im Fall des letzten Elements nirgends hin verweist:

# Datensatz	KontoNr	Filiale	Kontostand	Verweis
record 0	K-1234	Graz	450	record 1
record 1	K-2345	Wien	750	record 2
record 2	K-9022	Salzburg	248	record 3
record 3	K-3748	Graz	876	record 4
record 4	K-8473	Linz	230	record 5
record 5	K-9876	Klagenfurt	345	<leer>

6.1.1 Statisches Hashing

Statisches Hashing ist eine spezielle Form der Dateiorganisation. Dabei werden die Datensätze in 'Buckets' gespeichert, welche danach eine konstante Zugriffszeit ermöglicht. Ein Bucket ist eine Speichereinheit welche aus mehreren Blöcken auf der Festplatte besteht und Datensätze mit bestimmten Suchschlüsseln beinhaltet. Die Schlüssel muss dabei jedoch nicht die selben sein. Das funktioniert, da eine Hashfunktion aus unterschiedlichen Inputs den gleichen Output generieren kann. Ein sehr einfaches Beispiel einer solchen Hashfunktion wäre Modulo. Da diese nur den Rest, jedoch nicht das Ergebnis speichert, kann es trotz unterschiedlicher Operation zum gleichen Rest kommen, welche dadurch gebündelt werden.
Das praktische an einer solchen Hashfunktion ist, dass das Hashing eine konstante Zeit benötigt und so unabhängig von der Position die gleiche Zeit benötigt. Bei sehr großen Datensätzen kann eine Hashfunktion etwas länger als konstante Zeit benötigen da der Algorithmus danach immer noch eine große Menge an Daten durchsuchen muss (Aber in keinster Weise die gesamte.) Idealerweise sollte eine Hashfunktion die gleiche Menge an Werten in alle Buckets verteilen und diese so zufällig wie möglich zuweisen. Also sollte bei 10 Buckets nicht einer die Hälfte aller Werte besitzen und auch nicht ab einem gewissen Wert einem dieser Buckets mehr zuweisen.
Die Performance einer Hashfunktion muss jedoch auch immer abwägen wie viel Rechenleistung pro Hash nötig ist, da der zwar konstante Wert trotzdem einen größeren Aufwand als nötig bedeuten kann.

6.1.1.1 Bucket Overflow

Ein Bucket kann jedoch auch voll werden, in welchem Fall ein Bucket Overflow entsteht. Das kann mehrere Gründe haben: Entweder es gibt zu wenige Buckets oder es besteht eine ungleichmäßige Verteilung der Buckets. Während ein einfacher Weg wäre die Größe oder Menge der Buckets zu erhöhen, ist in der Regel eine ungleiche Verteilung der Grund für einen Overflow, weshalb man versuchen sollte einen anderen Algorithmus zu verwenden.

6.1.1.1.1 Closed Addressing (Overflow Chaining)

Ein Overflow kann jedoch nie komplett vermieden werden, weshalb man Overflow Chaining betreiben kann. Hierbei wird, falls ein Bucket einen Overflow erlebt, der letzte Wert durch einen Pointer auf einen neuen Pointer ersetzt, welcher wiederum neue Werte enthält. Dieser neue Bucket hat jedoch immer noch den gleichen Hashwert als der ursprüngliche Bucketm weshalb er als Closed Addressing bezeichnet wird.

6.1.1.1.2 Open Addressing

Alternativ dazu besteht das Open Addressing bei der die Menge der Buckets fix ist und bei einem Overflow wird dieser in einen anderen, bereits existierenden Bucket überführt. Diese Strategie findet jedoch in Datenbanken nur selten Anwendung da das Löschen von Einträgen sich als außerst schwierig gestaltet.

6.1.1.2 Beispiel:

Ein Beispiel für statisches Hashing wäre die Organisation einer Relation mit dem Standort als String als Suchschlüssel. Dabei wird der Modulo 10 der Summe der numerischen Werte der Buchstaben als Bucket verwendet. So können gleichzeitig 10 Buckets existieren wodurch idealerweise nur ein zehntel der Datenbank durchsucht werden muss. Wenn der Suchschlüssel also `Linz` ist, muss man `l + i + n + z -> 12 + 9 + 14 + 26 = 61` rechnen, dieses Ergebnis durch 10 dividieren und dessen rest nehmen: `61 % 10 = 1`. Also würde Linz in den Bucket Nummer 1 gehen.

6.2 Dynamisches Hashing

Eine Alternative zum statischen Hashing ist das dynamische Hashing. Der Umstand, dass es sich verändern kann hat einen wichtigen Vorteil, da die Größe einer Datenbank sich ständig verändert. Am Anfang eines Lebens einer Datenbank ist diese noch sehr klein und wird eventuell nie größer, die Veränderung wenn man viele Daten hat ist jedoch extrem aufwändig. Beim dynamischen Hashing kann man die Menge der Buckets dynamisch bestimmen. Dabei gibt es wiederum zwei Wege dies zu vollbringen: Das reguläre dynamische Hashing oder das erweiterbare Hashing, was eine spezielle Form des dynamischen Hashings darstellt.

6.2.1 Erweiterbares Hashing

Bei dem erweiterbaren Hashing berechnet die Funktion den Hashwert für eine sehr große Menge an Buckets. Man kann zum Beispiel eine 32-Bit Zahl verwenden um so 4 Milliarden Buckets zu definieren. Bei dem Hashing wird jedoch nur ein Teil des Hashwerts verwendet. So kann man nur den niederwertigsten oder höchstwertigsten Bit verwenden um den Bucket zu wählen. So kann man genau definieren welche Buckets Verwendung finden, man kann die Menge jedoch stets erhöhen. Bei der Abspeicherung wird anhand der Anforderungen ein Teil des Hashwerts verworfen und danach anhand einer Tabelle ein Bucket zugewiesen. Wenn nun jedoch ein Bucket voll ist, kann man diesen in zwei Teile spalten (Bucket Split). Danach werden die Inhalte neu gehasht und in die neuen Buckets verteilt. Was man hierbei jedoch beacthen muss, ist, dass die globale Tiefe (Die Tiefe der Verteilungstabelle) nie niedriger sein darf als die eines lokalen Buckets (Die Tiefe der Buckets selbst)
Wenn die Werte in der Datenbank sich verringern, können Buckets auch wieder verschmolzen werden. Dazu müssen zwei Buckets den gleichen um 1 verringerten Bitwert haben. (Also können zwei Buckets mit den Werten 11 und 01 verschmolzen werden, aber nicht zwei Buckets mit den Werten 11 und 10) Zusätzlich dürfen sich in beiden Buckets keine Werte mehr befinden. Danach kann man die Buckets verbinden und die Pointer neu verteilen. Wenn dieser verschmolzene Bucket als einziger noch die größere Tiefe hatte, kann man danach auch die globale Tiefe um eins verringern.
Der große Vorteil dieses Ansatzes ist, dass Datenabruf relativ günstig ist, da die Menge der Buckets stets eine akzeptable Größe haben. Man muss auch die Datenbank nie neu verteilen, da nur lokal neue Werte generiert werden müssen. Es kann jedoch passieren, dass viele Werte in einen Bucket gegeben werden, was die lokale Größe der Buckets negativ beeinflussen kann. Zusätzlich ist die Größe der Buckets statisch wodurch man selbst bei einer kleinen nötigen Vergrößerung diesen Bucket Split durchführen muss.

6.2.1.1 Beispiel:

Wenn man also beispielweise 7 Werte hat (16, 4, 24, 6, 22, 10, 31) und die globale Tiefe 1 ist, wird nur das LSB für die Buckets herangezogen. Es wird angenommen, dass ein Bucket nur 3 Werte gleichzeitig aufnehmend kann. Die Bitwerte dieser Zahlen sind also (10000, 00100, 11000, 00110, 10110, 01010, 11111) und es gibt zwei Buckets, einen mit dem Wert 0 und einen mit dem Wert 1. Wir sehen, dass bereits die ersten 4 Werte zu einem Bucket Overflow führen. Aus diesem Grund wird die lokale Tiefe des Bucket 0 auf 2 erhöht. Da die globale Tiefe noch 1 ist, muss diese auch auf 2 erhöht werden. Aus diesem Grund hat die Tabelle nun 4 Möglichkeiten zur Verteilung: 00, 01, 10 und 11. Die drei Werte welche bereits in dem 0 Bucket waren müssen nun neu gehasht werden, wobei jeweils die letzten zwei Bit zur Evaluierung herangezogen werden. Da alle drei wiederum 00 als letzte Bits haben werden sie in den Bucket 00 gegeben. Das vierte Element hat jedoch den Wert 10 und kommt deshalb auch in den Bucket 10. Das selbe passiert mit den nächsten zwei Werten. Nur der siebte Wert 11111 hat eine 1 am Ende und kommt deshalb in einen anderen Bucket. Da die lokale Tiefe des 1 Buckets immer noch 1 ist, verweisen beide Buckets 11 und 01 auf den selben Bucket.

6.2.2 Katalog

Zusätzlich zur Speicherung der Daten selbst gibt es den Datenbankkatalog. Dieser speichert stets Metainformation der Datenbank wie den Namen der Relation, die Attribute und Typen jeder Relation sowie Integritätsbedingungen. Gleichzeitig verwaltet es auch die Berechtigungen der einzelnen Benutzer um zu kontrollieren wer die Datenbank verändern oder neue Rechte vergeben darf. Es gibt zusätzlich noch weitere Information wiie die Anzahl der Attribute sowie ein Cache für oft verwendete Werte.

Die physische Dateiorganisation wird auch hier gespeichert wie den Speicherform der Relation sowie deren phsysischer Speicherort und die Adresse im Speicher des ersten Blocks.
Der Katalog selbst hat auch ein spezielles Schema um diesen für effizienten Zugriff zu optimieren. So werden die Relationen, Attribute Benutzer sowie Index und View in einem speziellen Format abgespeichert: `RELATION-METADATA (relation_name, nr_of_attributers, storage_organization, location)`

7 Index

Ein Index ermöglicht einen beschleunigten Zugriff auf eine Datenbank, änhlich zum Verzeichnis eines Buches. Eine Index Datei besteht in der Regel aus der Form des Indexeintrags, einem Schlüssel welcher zum Finden benötigt wird, sowie ein Pointer auf den Datensatz selbst. Es ist wichtig, dass ein Index stets kleiner sein muss, als die Datenbank selbst, da man sonst keinen Performancegewinn hat.
In einem Index können mehrere Arten von Anfragen geschehen: Eine Punktanfrage sucht anhand eines spezifischen Schlüssels (z.B. `id=6543`). Eine Mehrpunktanfrage und Bereichsanfrage [TODO ANFRAGE]

7.1 Indextypen

Indextypen unterscheidet man nach verschiedenen Kriterien:

- Die Ordnung der Index Datei
  - Clustering Index
  - Non-Clustering Index
- Die Art der Indexeinträge
  - Sparse Index
  - Dense Index

Indextypen basieren auf einer Kombination dieser zwei Kriterien. Dabei sind jedoch einige Typen nicht möglich oder kommen nur selten vor. Ein Clustering Index ist zum Beispiel meistens sparse, während ein Non-Clustering Index *immer* dense ist.
Wahr für alle Indextypen ist, dass Punktanfragen signifikant schneller bearbeitet werden können. Dieser Fakt ist weniger wahr für Mehrpunktanfragen und Bereichsanfragen. Ein Index erzeugt jedoch auch Mehrkosten da bei jeder Änderung der Datenbank der Index angepasst werden muss.

7.1.1 Clustering Index

Bei einem Clustering Index ist sowohl die Index, als auch die Daten Datei, in einer nach dem Suchschlüssel sequentiell geordneter Form. Das macht sowohl Punkt- als auch Mehrpunkt- und Bereichsanfragen sehr effizient und lässt einen guten Nicht-Squentiellen Zugriff zu. Großer Nachteil ist, dass man *nur einen* Clustering Index pro Tabelle haben kann. Wenn man also einen Clustering Index auf eine Kontonummer hat, muss man für den Kontonamen einen Non-Clustering Index anfertigen. Wenn nur ein Clustering Index für eine Suche mit einem anderen Kriterium vorhanden ist, muss die gesamte Tabelle manuell durchsucht werden. Also sollte man einen Clustering Index für oft verwendete Werte verwenden und sonst Werte als Non-Clustering Index abspeichert.

7.1.2 Non-Clustering Index

Ein Non-Clustering Index ist in der Index Datei sequentiell geordnet, jedoch nicht bei den Daten selbst. So verweisen Pointer in dem Index zu [TODO Punktanfrage ja, anderes nein, falls keine großen Datenmengen eventuell]

7.1.2.1 Primär- Sekundärindex

In der Praxis unterscheidet man zwischen dem Primärindex und dem Sekundärindex. In der Regel bedeutet das nur, dass der Primärindex der Clustering Index ist und der Sekundärindex andere Non-Clustering Indexe sind.

7.1.3 Sparse Index

Ein Sparse Index hat einen Index Eintrag welcher auf mehrere Datensätze verweist. So kann man die Größe des Index verringern. Ein Sparse Index ist jedoch nur möglich wenn man einen Clustering Index hat, da ein Sparse Index nur funktioniert wenn der Verweis auf einen Eintrag sequentiell abarbeitbar ist. Ein Clustering Index kann jedoch auch Dense sein.

7.1.4 Dense Index

Im Dense Index hat jeder Index-Eintrag einen eindeutig zugewiesenen Ort in den Daten. Aus diesem Grund ist ein Dense Index bedeutend größer als ein Sparse Index, ist in der Regel jedoch kleiner als die Daten selbst da der Index ja nur die Pointer und nicht die Daten selbst beinhaltet. Aus diesem Grund muss ein Non-Clustering Index immer Dense sein. Ein Vorteil eines Dense Index ist, dass eventuell eine Datenbank gelesen werden kann, ohne dass direkt auf diese zugegriffen werden muss. Das nennt man dann einen 'Covering Index'.

7.2 Duplikate im Index

Wenn in einem Index Einträge doppelt vorhanden sind kann dessen Handhabung relativ schwer sein. Das ist vor allem im Falle von B^+ Baum Indexen der Fall (B^+ Bäume können anders als Binärbaume viele Kinder pro Knoten haben. (Siehe Datenstrukturen und Algorithmen))
Man kann damit jedoch umgehen indem man Mehrere Indexeinträge einführt. Zusätzlich kann es ein Problem bei Buckets sein, da dann ein Eintrag im Index zu mehreren möglichen Bucketeinträgen führt. Schließelich muss man den Suchschlüssel noch eindeutig machen. Dabei kann ein Tupel Identifier (TID) helfen um gleiche Datensätze eindeutig zu machen. Gleichzeitig kann man auch aus einer Kombination des TID und dem Bucket einen eindeutigen Key definieren.
In der Praxis werden jedoch doppelte Einträge als solche behandelt und diese auch zurückgegeben. Im Rahmen dieser Lehrveranstaltung wird jedoch angenommen, dass alle Datensätze immer eindeutig sind, da manche besprochenen Inhalte sonst nicht möglich sind.