Teil 4:

Transaktionen, Isolation, Speicherhierarchie, Abarbeitung

Transaktionen

Unter einer *Transaktion* versteht man eine Folge von SQL-Anweisungen, die als untrennbare Einheit ausgeführt werden. Transaktionen bestehen aus einer Folge von einzelnen SQL-Anweisungen, die aber nicht beliebig lange dauern sollte. Im speziellen ist es sehr unschön (um nicht zu sagen falsch), wenn man während einer Transaktion auf eine Benutzereingabe wartet.

Konkret erwartet man von einer Transaktion die ACID Eigenschaften.

Atomicity

Eine Transaktion muss eine untrennbare Einheit darstellen. Entweder es werden alle Teiloperationen ausgeführt oder gar keine.

Consistency

Nach dem Ende einer Transaktion müssen die Daten in einem konsistenten Zustand sein, das heißt, alle Integritätsbedingungen müssen erfüllt sein.

Isolation

Parallel laufende Transaktionen müssen voneinander isoliert sein. Eine Transaktion darf Änderungen einer anderen nicht abgeschlossenen Transaktion nicht sehen. Diese Eigenschaft wird auch Serialisierbarkeit genannt (siehe auch Isolationsstufen weiter unten).

Durability

Nach dem Abschluss einer Transaktion sind die Auswirkungen dauerhaft. Sie bleiben auch im Fall eines Systemfehlers (Stromausfall etc.) erhalten.

Transaktionsanweisungen in SQL

```
§ TransactionalStat =
                        "BEGIN" "TRANSACTION"
                      | "COMMIT" "TRANSACTION"
                      | "ROLLBACK" "TRANSACTION".
```

Manchmal ist TRANSACTION auch abgekürzt zu TRAN. Mit BEGIN TRANSACTION wird eine Transaktion gestartet. Es folgen beliebige andere SQL-Anweisungen. Die Transaktion wird mit COMMIT TRANSACTION abgeschlossen oder mit ROLLBACK TRANSACTION verworfen. Start und Ende treten also stets paarweise auf und bilden eine Klammer um die eingeschlossene Folge von SQL-Anweisungen.

Transaktionen sind (normalerweise) nicht schachtelbar. Es gibt aber SQL-Erweiterungen, die geschachtelte Transaktionen anbieten.

Isolationsstufen

Die in der ACID Eigenschaft 'Isolation' geforderte Serialisierbarkeit von Transaktionen führt streng genommen zu einer sequentiellen anstelle einer parallelen Ausführung. Man möchte aber kurze Antwortzeiten für parallel arbeitende Benutzer erzielen und die Effizienz eines RDBMS steigern, indem man Transaktionen parallel ausführt. Die Eigenschaft 'Isolation' wird dabei mehr oder weniger gut angenähert wie in folgenden vier SQL-92 Isolationsstufen beschrieben ist. Die dabei beobachtbaren Anomalien, die bei streng sequentieller Abarbeitung nicht auftreten, werden *Phänomene* genannt.

READ UNCOMMITTED: in manchen Ausnahmefällen ist es akzeptabel, wenn eine Transaktion eine Änderung einer anderen Transaktion sehen kann, bevor diese abgeschlossen ist. Man spricht in diesem Fall vom *dirty read*-Phänomen. Man beachte, dass einzelne Tupel auch in dieser Isolationsstufe immer als untrennbare Einheit gelesen oder geschrieben werden. Es kann also nicht passieren, dass man ein halb fertiges Tupel oder gar ein halb fertiges Attribut liest.

READ COMMITTED: wenn dirty-reads nicht akzeptabel sind, kann mit dieser strengeren Isolationsstufe gearbeitet werden. Eine Transaktion kann keine Änderungen einer anderen Transaktion lesen, solange diese nicht committed sind. Es kann aber in dieser Stufe immer noch folgendes Problem auftauchen: Eine Transaktion liest dieselben Daten mehrfach. Die Daten können in der Zwischenzeit von einer andern Transaktion geändert (oder gelöscht) und committed worden sein. Das Ergebnis des Lesens ist nicht garantiert wiederholbar. Man spricht vom *non repeatable read*-Phänomen. Wenn keine repeated reads verwendet werden, stellt diese Isolationsstufe eine gute Wahl dar, wird auch häufig in der Praxis verwendet und ist der Default bei vielen RDBMS.

REPEATABLE READ: Diese noch strengere Isolationsstufe garantiert, dass alle Daten, die in einer Transaktion einmal gelesen wurden, bei wiederholtem Lesen unverändert bleiben, also in der Zwischenzeit nicht von anderen Transaktionen geändert (oder gelöscht) und committed werden können. Es kann aber immer noch folgendes Problem auftauchen. Eine Transaktion führt eine SELECT-Anweisung aus und bekommt eine Tupelmenge als Ergebnis, die auch gelesen wird. Alle gelesenen Daten bleiben zwar unverändert, es könnte aber in der Zwischenzeit durch andere Transaktionen Daten hinzugefügt werden. Ein neuerliches Ausführen des gleichen SELECTs liefert dann möglicherweise eine größere Ergebnismenge. Man spricht in diesem Fall vom *phantom read*-Phänomen. Hinweis: in manchen RDBMS wird REPEATABLE READ gleichbedeutend mit SERIALIZABLE verwendet.

SERIALIZABLE: Diese strengste Isolationsstufe schließt alle oben genannten Phänomene aus und garantiert die Abarbeitung so, als würde eine sequentielle Bearbeitung erfolgen. Tatsächlich erfolgt die Abarbeitung aber (meist) parallel. Zur Aufrechterhaltung der logischen Trennung werden Locks (Sperren, siehe unten) verwendet. Da bei höheren Isolierungsebenen mehr Locks als bei niedrigeren verwendet werden müssen, können höhere Isolierungsebenen einerseits leichter zu Deadlocks führen und es wird andererseits ein geringeres Maß an Parallelität erreicht.

Ein RDBMS erlaubt das Setzen der Isolationsstufe z.B. mit folgendem Befehl

§ SetIsolationStat = "SET" "TRANSACTION" "ISOLATION" Const.

Locks und Deadlocks

Ein RDBMS verwendet üblicherweise Locks (=Sperren), um obige Isolationsstufen zu realisieren. Ein Lock ist eine Datenstruktur, die ausdrückt, dass eine bestimmte Ressource (Tabelle, Tupelbereich, Tupel, etc.) in einem bestimmten Sinn für andere Transaktionen gesperrt ist.

Ein *Deadlock* liegt vor, wenn zwei (oder mehr) Transaktionen wechselseitig aufeinander warten, so dass keine der Transaktionen einen Fortschritt erzielen kann. Je höher die Isolationsstufe ist, desto leichter können Deadlocks entstehen. Das liegt daran, dass bei höherer Isolation mehr Locks benötigt werden. Bei READ UNCOMMITTED werden z.B. Locks anderer Transaktionen generell ignoriert. Es kann keinen Deadlock geben. Bei READ COMMITTED werden Write-Locks auf geänderten Tupel beachtet. Bei REPEATABLE READ werden zusätzlich auch beim Lesen von Daten Read-Locks gesetzt und solche auch beachtet.

Beispiel für Deadlock zweier Transaktionen unter der Annahme der Isolierungsebene READ COMMITTED und 'Table Lock':

Transaktion1 beginnt.

Transaktion 1 verändert Tabelle A und sperrt damit Tabelle A für andere.

Transaktion 2 beginnt.

Transaktion 2 verändert Tabelle B und sperrt damit Tabelle B für andere.

Transaktion1 möchte Tabelle B ändern, muss aber wegen Sperre durch Transaktion 2 warten.

Transaktion 2 möchte Tabelle A ändern, muss aber wegen Sperre durch Transaktion 1 warten.

-- deadlock: beide Transaktionen warten wechselseitig aufeinander.

Das RDBMS wird eine der beiden Transaktionen als *deadlock victim* (Opfer) betrachten und abbrechen. Die abgebrochene Transaktion wird wie mit ROLLBACK rückgesetzt.

Abhilfe gegen Deadlocks in SQL

1. Eine Möglichkeit besteht darin, eine niedrigere Isolationsstufe zu wählen, sofern das für die Anwendung akzeptabel ist.

2. Eine zweite Möglichkeit besteht darin, die Reihenfolge der Ressourcenbelegung bei allen Transaktionen (wenn irgendwie möglich) gleich zu lassen. In obigem Beispiel müsste Transaktion 2 so umgestaltet werden, dass sie auch zuerst Tabelle A ändert und dann erst B. Dann muss Transaktion 2 zwar warten, aber Transaktion 1 wird nicht blockiert und es kommt zu keinem Deadlock.

3. Eine weitere Möglichkeit besteht darin, mit dem Auftreten von Deadlocks zu rechnen und die Anwendung so zu gestalten, dass ein automatischer Restart eines Deadlock-Victims erfolgt.

Da in SQL die Belegung der Ressourcen nicht bei BEGIN TRANSACTION angegeben wird, sondern dynamisch durch die Ausführung der Transaktionsschritte erfolgt, ist es nicht möglich, mit einer generellen Deadlock-Vermeidungsstrategie zu arbeiten. Die einzige Möglichkeit wäre die echt sequentielle Ausführung der Transaktionen, aber das wird als nicht praktikabel erachtet.

Lock Granularität

Generell sind die Details der Verwendung von Locks RDBMS abhängig und einer der Schlüssel für effizienten Mehrbenutzerbetrieb. Je nach *Granularität* der Sperren unterscheidet man prinzipiell zwischen folgenden Arten von Locks:

Global Lock: es wird die gesamte RDBMS gesperrt. z.B. für Einbenutzerbetrieb oder für Schemaänderungen.

Table Lock: es wird eine ganze Tabelle gesperrt.

Range Lock: es werden alle Tupel einer Tabelle in einem bestimmten Bereich (von, bis) gesperrt.

Page Lock: es werden alle Tupel einer Seite (=Speichereinheit auf der Festplatte) einer Tabelle gesperrt.

Row Lock: es wird genau ein Tupel gesperrt.

Unter *Lock Escalation* versteht man die automatische Änderung der verwendeten Locks während der Abarbeitung einer Transaktion von kleinerer zu größerer Granularität. Aus einer großen Zahl von Row-Locks könnte z.B. ein oder eine kleine Menge von Page-Lock werden oder aus mehreren Page-Locks könnte ein Table-Lock werden. Row Locks haben das kleinste Potential für Deadlocks, sind aber am aufwändigsten zu verwalten, weil sie in großer Anzahl vorkommen können.

Zweck eines Locks

Neben der Granularität unterscheidet man Locks auch hinsichtlich ihres Zweckes. Häufige Lock-Arten sind:

- *Read-Locks* (auch *shared locks* genannt). Diese erlauben das Lesen für andere Transaktionen nicht aber das Schreiben.
- *Write-Locks* (auch *exclusive locks* genannt). Diese sperren sowohl Lese- als auch Schreibzugriffe für alle anderen Transaktionen.

Snapshot Isolation

Manche Datenbanken (z.B. Postgres, Borland InterBase) verwenden an Stelle von Locks einen *Snapshot*-Mechanismus. Das bedeutet, dass man innerhalb einer Transaktion immer nur den Zustand der Daten wie zu Beginn der Transaktion sieht (daher auch "timestamp based Isolation" genannt). Änderungen, die innerhalb der Transaktion durchgeführt wurden, sind für keine Transaktion, auch nicht für die durchführende, sichtbar. Bei Datenbanken, die Locks verwenden, sind hingegen die Änderungen für die durchführende Transaktion sichtbar (Die Portierung von Anwendungen ist in solchen Fällen besonders heikel, woraus man folgern kann, dass Transaktionen nach Möglichkeit durchgeführte Änderungen nicht wieder auslesen sollten. Das vermeidet auch das Auftreten des non-repeatable-read Phänomens)

COMMIT bedeutet hier, dass alle Änderungen der Transaktion in die Datenbank geschrieben werden. Dabei wird nun geprüft, ob zwischen BEGIN und COMMIT eine Änderung der zu schreibenden Daten durch eine andere bereits abgeschlossene Transaktion stattgefunden hat. Wenn ja, gibt es einen Update-Konflikt und ein ROLLBACK. Das erste COMMIT gewinnt.

Langlaufende Transaktionen sind hier kein großes Problem, da weder Lese- noch Schreibzugriffe anderer Transaktionen dadurch blockiert werden. Durch das first-commit-wins Prinzip benachteiligt sich eine langlaufende Transaktion allenfalls selbst. Mit offenen alten Transaktionen kann auch auf historischen Daten gearbeitet werden. Nur Updates auf alten Daten würden in vielen Fällen zu Update-Konflikten führen.

Snapshot Isolation erfüllt alle Anforderungen der Isolationsstufe SERIALIZABLE ohne das Lesen jemals zu blockieren. Snapshot-Isolation gilt daher als moderner Ansatz, hat sich aber noch nicht auf breiter Front gegenüber Lock-based Isolation durchgesetzt. Ein Nachteil von Snapshot Isolation besteht darin, dass es nicht möglich ist eine Sequentialisierung zu erzwingen, was zB bei der Realisierung eines Zählers (etwa für die Vergabe einer fortlaufenden Artikelnummer) erforderlich ist. Hier müssten Update-Konflikte eingeplant und ein Restart im Anwendungsprogramm vorgesehen sein.

Optimistic Update

Da Transaktionen kurz sein sollen und im speziellen keine Benutzereingaben umfassen sollten, stellt sich die Frage, wie man das gegenseitige Überschreiben von Daten durch mehrere parallel arbeitende Benutzer vermeiden soll.

Eine häufig verwendete Strategie dafür ist das so genannte *Opimistic Update*. Dabei kann jeder Benutzer Daten lesen und bearbeiten, ohne dass Sperren gesetzt werden. Erst beim Schreiben einer Änderung wird überprüft, ob ein anderer Benutzer in der Zwischenzeit die gleichen Daten ebenfalls geändert hat. Dieser Ansatz ist ähnlich zu Snapshot Isolation, kann aber unabhängig von der verwendeten Isolationsimplementierung und ohne Verwendung von langlaufenden Transaktionen verwendet werden. Das Prinzip ist wie folgt:

1. Lesen eines Datensatzes unter Angabe des Primärschlüssels und Abspeichern des Tupels in Programmvariablen.
2. Bearbeiten des gelesenen Datensatzes durch Anwendungsprogramm und/oder Benutzer. Die ursprünglich gelesenen Daten bleiben dabei erhalten. Die geänderten Daten werden in zusätzlichen Programmvariablen gespeichert.

3. Schreiben des Datensatzes unter Verwendung des Primärschlüssels und zusätzlichem Vergleich des aktuellen Tupelwertes in der DB mit den ursprünglich gelesenen Werten. Falls ein Update-Count von 0 resultiert (Update-Count = Anzahl der geänderten Tupel), wurde der Datensatz in der Zwischenzeit gelöscht oder verändert. Ein Update-Konflikt wurde erkannt. Sonst ist der (optimistisch) erwartete Erfolgsfall eingetreten.

Beim Vergleich des aktuellen DB-Tupelwertes mit den ursprünglich gelesenen Daten sind verschiedene Ansätze möglich.

- Einführen eines Attributes mit einem Update-Zähler, der bei jedem Update um eins erhöht wird. Der Vergleich kann sich auf den Vergleich des Zählers beschränken. Führt zu kompakten Kommandos.
- Statt eines Zählers könnte auch ein Attribut mit einem Änderungszeitpunkt eingeführt werden. Diese Vorgangsweise ist aber fragil bei Änderung der Systemzeit eines Computers oder bei zufälligen Gleichzeitigkeiten von Operationen mehrere Benutzer und daher nicht empfehlenswert.
- Wertevergleich aller Attribute. Vermeidet zusätzliches Attribut für Zähler. Maximale Kommandolänge.
- Wertevergleich aller geänderten Attribute. Vermeidet zusätzliches Attribut für Zähler und ermöglicht paralleles Bearbeiten verschiedener Attribute durch verschiedene Benutzer, also weniger Update-Konflikte. Das Kommando wird dynamisch erstellt und die Kommandolänge ist proportional zur Anzahl der geänderten Attribute. Wenn es Abhängigkeiten zwischen den Attributen gibt, ist dieser Ansatz allerdings nicht möglich.

Beispiel für Optimistic Update mit Update-Zähler bei Änderung einer Artikelbezeichnung:

```
UPDATE Artikel SET bezeichnung = 'neueBezeichnung', cnt = cnt + 1
WHERE artnr = 1234
  AND cnt = Zählerstand beim Einlesen
```

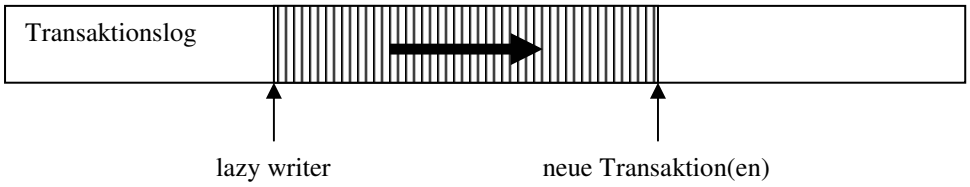
Beispiel für Optimistic Update mit Wertevergleich der geänderten Attribute bei Änderung einer Artikelbezeichnung:

```
UPDATE Artikel SET bezeichnung = 'neueBezeichnung'
WHERE artnr = 1234
  AND bezeichnung = eingelesene Bezeichnung
```

Transaktionslog

Um eine effiziente Abarbeitung von Transaktionen zu ermöglichen, verwenden die meisten RDBMS (egal wie Isolation implementiert ist) eine persistente Hilfsdatenstruktur, das *Transaktionslog*. Alle Aktionen einer Transaktion werden zunächst protokolliert und sequentiell in eine Log-Datei geschrieben. Änderungen an Tabellen erfolgen zunächst nur im Hauptspeicher. Aus dem Transaktionslog kann exakt rekonstruiert werden, welche Änderungen an welchen Relationen durchgeführt wurden. Dadurch kann im Fall eines Abbruchs (Rollback) die Auswirkung einer Transaktion wieder zurückgesetzt werden. Ein COMMIT wird ebenfalls zuerst nur in das Transaktionslog geschrieben. Im Falle eines Systemfehlers (Stromausfall, etc.) kann beim Neustart des RDBMS durch Analyse des Transaktionslogs ein erneutes Abarbeiten der Transaktion erfolgen (REDO) und die D-Eigenschaft von Transaktionen ist garantiert.

Damit das Transaktionslog nicht beliebig lang wird, wird es von der anderen Seite her von einem parallel laufenden Prozess (lazy writer) abgearbeitet und die Änderungen von abgeschlossenen Transaktionen werden in die eigentlichen Datentabellen übernommen und dort dauerhaft abgespeichert. Das Log bekommt dadurch die Struktur eines zyklischen Puffers, der an einer Seite wächst und auf der anderen Seite schrumpft. Das Transaktionslog stellt eine eigene Datei dar, die sequentiell geschrieben wird. Zur Maximierung des Durchsatzes eines RDBMS sollte daher eine eigene Disk (mit eigenem Diskcontroller) verwendet werden. Dadurch bleibt der Schreib-/Lesekopf immer an der Schreibposition des Transaktionslogs und man vermeidet unnötige Wartezeiten (siehe Festplatten).



Speicherhierachie

Die Daten einer oder mehrerer Tabellen werden meist auf eine oder mehrere Dateien eines Filesystems (FAT, FAT32, NTFS, HPFS, rfs, etc.) abgebildet, das Transaktionslog ebenso. Daneben gibt es auch (vereinzelt noch) RDBMS, die direkt auf die Festplatte zugreifen, ohne ein dazwischenliegendes Dateisystem zu verwenden. Die Daten werden aber letztlich immer auf *Festplatten* gespeichert, weil heutzutage nur damit eine dauerhafte und kostengünstige Speicherung großer Datenmengen (mehrere GB bis TB) möglich ist. Bei Festplatten kann man mit durchschnittlichen Zugriffszeiten im 10 msec-Bereich rechnen.

Neben den Festplatten werden auch noch *Magnetbänder* verwendet, aber nicht für den eigentlichen Betrieb, sondern für Archivierung von Daten und für Sicherungskopien. Durch die große Kapazität heutiger Festplatten (100 GB und mehr) und die Verwendung mehrerer Festplatten für eine Datenbank ist oft die Verwendung von Bandrobotern erforderlich. Diese wechseln automatisch bei Bedarf die Bänder und erlauben somit eine noch größere Speicherkapazität (TB) ohne organisatorisches Chaos. Die Zugriffszeit bei Bändern liegt infolge des sequentiellen Zugriffs und der eventuellen Notwendigkeit von Bandwechseln im Sekunden- bzw. Minutenbereich, ist also mindestens 1000 mal langsamer als bei Festplatten.

Zwischen RDBMS und Festplatten können zur Beschleunigung von Lese-/Schreiboperationen auch noch *RAM*-Speicher eingesetzt werden, die wiederum Faktor 100000 schneller sind als Platten. Die RAM-Speicher können in der Festplatte integriert sein und/oder vom Betriebssystem eines Rechners im Hauptspeicher angelegt werden.

Das RDBMS selbst liegt im Hauptspeicher eines Rechners, der ebenfalls aus RAM-Bausteinen besteht. Die CPU ist aber viel schneller als der Hauptspeicher, weshalb heutige Computer mit einem schnellen Zwischenspeicher ausgestattet sind (*Cache*). Dieser ist ca 128 KB bis 2MB groß. Bei vielen CPUs kommt davor ein weiterer noch schnellerer Cache-Speicher, der direkt im CPU-Chip integriert ist und ca 16 bis 32KB groß ist. Das schnellste Speichermedium ist das *Register*, auf das mit der vollen Taktgeschwindigkeit einer CPU zugegriffen wird (derzeit bis etwa 3 GHz).

Die Beachtung dieser Speicherhierarchie ist der Schlüssel für effizienten Umgang mit großen Datenmengen.

Speicherart	**typische Zugriffszeit**	**Dauerhaft**
Band	Sekunden bis Minuten	ja
Festplatte	10 Millisekunden	ja
RAM	0.1 Mikrosekunden	nein
Cache (L1, L2)	20 Nanosekunden	nein
Register	1 Nanosekunde	nein

Festplatten

Die Zugriffszeit auf eine Festplatte folgt aus deren Konstruktion und besteht aus:

- Seek time: Zeit für Bewegung des Schreib-/Lesekopfes zu einer bestimmten Spur (größter Anteil).
- Wartezeit: Zeit bis der gewünschte Sektor zum Kopf kommt. Im Durchschnitt die halbe Umdrehungszeit.
- Lesezeit: Zeit zum Lesen eines Sektors (kleinster Anteil, Sektorgröße ca. von 512B bis 4KB).

Zur Beschleunigung der Zugriffszeit und Reduktion der Ausfallsgefahr von Festplatten wurden verschiedene Techniken entwickelt. Eine (theoretische) Möglichkeit wäre, ausfallsarme und schnelle Festplatten zu entwickeln und einzusetzen. Diese Lösung ist aber teuer und kann letztlich gewisse physikalische Grenzen nicht überschreiten. Eine andere, populärere Möglichkeit besteht in der Verwendung von mehreren billigen Festplatten, die so zusammenwirken, dass eine gewisse Ausfallssicherheit, eine Beschleunigung von Lese- und/oder Schreiboperationen und/oder eine Erhöhung der Kapazität erzielt wird. Dieser Ansatz heißt RAID (Redundant Array of Inexpensive Disks, manchmal auch Redundant Array of Independent Disks genannt) und wird heute allgemein für Produktionsdatenbanken verwendet. RAID ist ein Sammelbegriff, der wieder in verschiedene konkrete Architekturen (RAID Levels) unterteilt wird, von denen die wichtigsten im Folgenden angeführt sind. Man beachte, dass RAID kein Teil des RDBMS selbst ist, sondern vom verwendeten Computersystem (Hardware ev. plus Betriebssystem) bereitgestellt wird. Bei der Installation und Konfiguration eines RDBMS ist es aber wichtig diese Begriffe zu kennen, damit die richtigen Entscheidungen getroffen werden können.

RAID-0 oder *Striping*. Hier wird ein Datenbestand systematisch und ohne Redundanz auf mehrere Platten aufgeteilt (Stripe Set). Bei drei Platten könnte zum Beispiel Sektor N auf Platte A, Sektor N+1 auf Platte B und Sektor N+2 auf Platte C sein. Sequentielles Lesen und Schreiben wird dadurch etwa drei mal so schnell sein als mit einer Platte wenn die drei Platten unabhängig voneinander operieren. Diese Technik erhöht aber gleichzeitig die Ausfallsgefahr um den Faktor drei.

RAID-1 oder *Mirroring*. Hier wird ein Datenbestand auf einer (oder mehreren) anderen Platte(n) gespiegelt, also redundant mitgeführt. Fällt eine Platte aus, kann auf der anderen weitergearbeitet werden. Austausch einer Platte im laufenden Betrieb (Hot Swap) ist prinzipiell möglich. Beim Lesen kann eine Beschleunigung wie bei RAID-0 erzielt werden. Beim Schreiben kann eine Verzögerung eintreten, weil eine Schreiboperation erst abgeschlossen ist, wenn sie auf allen beteiligten Platten stattgefunden hat. Man wartet auf das Maximum der Schreibzeiten. Gute Wahl für das Transaktionslog wegen Ausfallssicherheit.

RAID-10 (oder 01, oder 0+1, oder 1+0) oder *Striping plus Mirroring*. Kombination aus beiden. Ein Stripe-Set wird vollständig gespiegelt. Hohe Lese- Schreibleistung inkl. Ausfallssicherheit aber um den Preis der nochmaligen Verdoppelung der Plattenanzahl. Beste Wahl für das Transaktionslog von High-End Datenbanken.

RAID-5 oder *Striping with distributed Parity*. Hier verwendet man Striping und fügt Redundanz in Form von Parity-Information hinzu. Man verwendet mindestens 3 Platten A, B und C, von denen jeweils zwei mit Striping arbeiten. Zwei im Sinn von Striping zusammengehörige Sektoren (z.B. Sektor N und N+1) werden auf zwei verschiedene Platten (z.B. A und B) geschrieben. Auf die dritte (C) wird Paritätsinformation geschrieben, mit deren Hilfe bei Ausfall einer der beiden Platten (A, B) die fehlende Information rekonstruiert werden kann. Es ist keine feste Platte nur für Parity zuständig, sondern es werden alle Platten abwechselnd mit Daten und Parity beladen (distributed parity). Die Paritätsinformation wird durch ein bitweises XOR gebildet, dass die Eigenschaft hat, dass: (C = A XOR B) => ((B = C XOR A) & (A = B XOR C)). Man überprüft das leicht mit einer Wahrheitstabelle. RAID-5 erlaubt damit schnelles Lesen wie bei Striping. Beim Schreiben ist allerdings Zusatzaufwand erforderlich, weil mit jedem geschriebenen Sektor auch der Paritysektor geschrieben werden muss und dazu entweder das Lesen des zugehörigen Stripe-Sektors oder (besser, spart einmal seek time) das Lesen des alten Paritysektors und des alten Datensektors erforderlich ist. Bei drei Platten kann eine ausfallen, ohne dass der Betrieb unterbrochen wird. Hot Swap ist prinzipiell möglich. RAID-5 wird häufig zur Speicherung der Tabellen eines RDBMS verwendet.

Kommando-Abarbeitung

Ein RDBMS basiert auf einem Programm, das schrittweise ausgeführt wird und letztlich mit den elementaren Maschinenbefehlen auskommen muss. Mengenoperationen, Joins etc. sind nicht direkt in Hardware verfügbar. Das RDBMS muss also ein SQL-Kommando so umwandeln, dass es mit den

verfügbaren Primitivoperationen ausgeführt werden kann. Diese Umwandlung erledigt einerseits der Query-Compiler und andererseits der Executor. Der Compiler bekommt ein SQL-Kommando als Eingabe und liefert einen Ausführungsplan aus Ausgabe. Der Ausführungsplan ist eine Datenstruktur (Operator Tree), die genau beschreibt, welche DB-Operationen konkret auszuführen sind. Die Ausführung übernimmt der Query-Executor, der einen Interpreter für Ausführungspläne darstellt. Bei der Erstellung des Ausführungsplanes ist darauf zu achten, dass der Executor rasch zu einem Ergebnis kommt. Oft gibt es mehrere Möglichkeiten, die Einzeloperationen anzuordnen, dann soll der Compiler die effizienteste Form wählen.

Der Aufwand für die Übersetzung und vor allem für die Optimierungen ist nicht zu vernachlässigen, daher die Trennung zwischen Compiler und Executor. Kommandos, die schon einmal optimiert wurden, werden nach Möglichkeit nicht jedesmals neu übersetzt, sondern der Ausführungsplan wird in einem Befehls-Zwischenspeicher (Query Cache) abgelegt und nach Möglichkeit wiederverwendet.

Der Zusatzaufwand, den der Executor zum Interpretieren des Ausführungsplanes aufweist, ist vernachlässigbar gegenüber den DB-Operationen, die er ausführt. Daher würde eine Übersetzung des Ausführungsplanes in die native Maschinensprache eines Rechners wenig Vorteile bringen. Sie würde aber die Portabilität eines RDBMS auf andere Hardwareplattformen erschweren.

Ein Beispiel soll das Potential für Optimierungen veranschaulichen. Betrachten wir folgende Query:

```
SELECT * FROM Lieferant L, Artikel A
WHERE L.lifnr = A.lifnr AND A.artnr = 42
```

Eine naive Ausführungsstrategie würde im Sinne der Relationenalgebra so vorgehen:

1. bilde das Produkt von L und A.
2. selektiere alle gültigen Tupel gemäß WHERE-Bedingung.

Das Problem dabei ist, dass das Produkt sehr groß werden kann. Nehmen wir 1.000 Lieferanten und 50.000 Artikel und wir bekommen 50.000.000 Tupel, die in Schritt 1 in eine Hilfstabelle geschrieben werden müssen. In Schritt 2 müssen alle 50.000.000 Tupel gelesen und auf die WHERE-Bedingung geprüft werden.

Eine bessere Strategie wäre die Verzahnung der Produktbildung mit dem Prüfen der WHERE-Bedingung wie in folgendem Algorithmus skizziert.

```
for each l IN L do
      for each a in A do
            sei t das Tupel aus a und l
            if (t erfüllt WHERE-Bedingung) then
                  füge t zu Ergebnis hinzu
            end if
      end for
end for
```

Hier bekommt man immer noch 50.000.000 Prüfungen der where-Bedingung, aber das Schreiben der Hilfstabelle entfällt.

Weitere Verbesserungen sind notwendig, damit man brauchbare Antwortzeiten erhält. Naheliegend ist die Ausnützung einer Sortierung, die man sicher auch bei manueller Suche in einem Karteikasten anwenden würde. Nehmen wir an, `A` ist nach `artnr` sortiert und `L` nach `lifnr`. Die Artikelnummer sei der Primärschlüssel, also eindeutig. Dann ergibt sich folgender Algorithmus:

```
for each l IN L do
      sei a ein Tupel aus A mit artnr = 42
      sei t das Tupel aus a und l
      if (t erfüllt join-Bedingung) then
            füge t zu Ergebnis hinzu
      end if
end for
```

Das Finden von `a` erfolgt bei binärer Suche mit logarithmischem Aufwand, d.h bei 50.000 Artikeln sind nur $\log_2 50000 = 16$ Schritte erforderlich (tatsächlicher Aufwand siehe B-Bäume). Der Gesamtaufwand setzt

sich zusammen aus 1000 Durchläufen durch die äußere Schleife, wobei jeweils `a` gesucht und die join-Bedingung getestet wird. Das ist sehr viel schneller als die vorherige Lösung, aber immer noch nicht optimal.

Wir können beobachten, dass das Suchen nach `a` in jedem Schleifendurchlauf gleich ist, also kann man den Aufwand reduzieren, indem man diesen konstanten Teil aus der Schleife herauszieht und erhält:

```
sei a ein Tupel aus A mit artnr = 42
for each l IN L do
      sei t das Tupel aus a und l
      if (t erfüllt join-Bedingung) then
            füge t zu Ergebnis hinzu
      end if
end for
```

Auch das kann noch weiter verbessert werden, wenn wir berücksichtigen, dass auch `L` sortiert ist. Es ist nicht notwendig, sequentiell alle Tupel aus `L` durchzupropieren, sondern man kann direkt aus `a` das entsprechende `a.lifnr` verwenden und in `L` danach suchen. Damit kommen wir zur endgültigen Lösung, die auch die Überprüfung der Existenz von `a` durchführt (wurde oben ignoriert).

```
sei a ein Tupel aus A mit artnr = 42
if (a existiert) then
      sei l ein Tupel aus L mit l.lifnr = a.lifnr
      if (l existiert) then
            füge das Tupel aus a und l zum Ergebnis hinzu
      end if
end if
```

Durch Ausnützung der Sortierung von `L` und `A` erhalten wir einen sehr effizienten Ausführungsplan, der aus log(|A|) + log(|L|) Schritten besteht, anstelle von |A| * |L|. Erst durch diese Art von Optimierung sind RDBMS in der Praxis einsetzbar.

Ein RDBMS kann tatsächlich von Sortierung Gebrauch machen, wenn die Primärschlüsselattribute verwendet werden. Jeder Primärschlüssel stellt ein Sortierkriterium dar. In vielen Fällen ist es aber wichtig, dass auch nach anderen Attributen effizient gesucht werden kann. In diesem Fall muss man explizit einen sogenannten *Index* erzeugen.

Index

SQL erlaubt die Erstellung eines Index durch die CREATE INDEX-Anweisung.

§ CreateIndexStat = "CREATE" ["UNIQUE"] "INDEX" name "ON" TableName "(" AttrList ")".

Mit der Option UNIQUE kann ausgedrückt werden, dass die Attributwerte eindeutig sein müssen. Ein Beispiel für einen UNIQUE INDEX ist der Primärschlüssel einer Tabelle. Dafür wird vom RDBMS normalerweise automatisch ein entsprechender Index angelegt.

Ein Index wird von einem RDBMS auch verwendet, um die ORDER BY Klausel auszuwerten. Wenn ein Index vorhanden ist, muss nicht extra sortiert werden, sonst schon. Da ORDER BY sowohl aufsteigendes als auch absteigendes Sortieren erlaubt, gibt es für einen Index auch die Möglichkeit ASC (default) oder DESC als Option nach einem Attributnamen anzugeben (nicht in obiger Grammatik).

Beispielindex zum effizienten Suchen nach Artikel.lifnr:

```
CREATE INDEX Artikel_Lifnr ON Artikel(lifnr)
```

Ein Index kann durch die DROP INDEX-Anweisung wieder entfernt werden.

§ DropIndexStat = "DROP" "INDEX" name.

zum Beispiel:

```
DROP INDEX Artikel_Lifnr
```

B-Bäume

Ein Index muss effizientes Suchen aber auch effizientes Einfügen und Löschen ermöglichen. Die dafür meistens verwendete Datenstruktur ist der B-Baum (Bayer, McCreight 1972) (nach einem der Erfinder 'Bayer' benannt, nicht binär), der in verschiedenen Variationen existiert. In jedem Fall handelt es sich um einen ausgeglichenen Mehrweg-Suchbaum, der sowohl Suchen als auch Einfügen und Löschen mit logarithmischem Aufwand ermöglicht. Zum Unterschied von einem Binärbaum wird beim B-Baum garantiert, dass keine Entartung (etwa durch sortiertes Einfügen) auftreten kann. Meist wird er in Form des B^+-Baumes verwendet, der sich dadurch auszeichnet, dass er die geringstmögliche Höhe aufweist.

Prinzip: Ein B-Baum ist immer ein Mehrweg-Baum, d.h. er hat nicht wie ein Binärbaum max. 2 Nachfolger, sondern 2**N*, wobei *N* so gewählt wird, dass ein ganzer Block (Sektor, Teil der Festplatte oder Vielfaches davon) mit einem Knoten gefüllt wird. *N* heißt der *Grad* des B-Baums. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge. Jeder Knoten (außer der Wurzel) enthält minimal *N* und maximal 2**N* Einträge.

Ein B^+-Baum (alias *hohler* Baum) unterscheidet zwischen Blattknoten und inneren Knoten. Die inneren Knoten verweisen auf Unterknoten und bilden die Sortierung. Die Blattknoten bilden Suchschlüssel auf Tupel ab, enthalten also die eigentlichen Daten des Index. Durch die Entfernung der Tupelreferenzen in den inneren Knoten kann der Grad des Baumes maximiert werden, wodurch dessen Höhe minimiert wird. B^+-Bäume haben sich in RDBMS weitgehend durchgesetzt.

Einfügen

Für jedes in die Relation eingefügte Tupel wird zunächst im Index nach dem relevanten Blattknoten (Einfügeposition) gesucht. Dieser Vorgang durchläuft den Baum von der Wurzel bis zum Blatt und die Anzahl der Suchschritte ist damit von der Höhe des Baumes abhängig. Falls Platz im Knoten mit der Einfügeposition vorhanden ist, wird ein Paar (Suchbegriff, Tupelreferenz) eingefügt. Wenn der Knoten mit der Einfügeposition schon voll ist (Anzahl Einträge ist 2**N*), erfolgt ein *Split*. Der volle Knoten wird dabei aufgeteilt auf zwei halb volle (jeweils *N* Einträge) und der neue Eintrag wird in einen der beiden gesplitteten Knoten hinzugefügt (*N*+1 Einträge). Im übergeordneten Knoten wird der neue Knoten *einsortiert* indem eine zusätzliche Verzweigung (Suchbegriff, Knotenreferenz) eingetragen wird. Der übergeordnete Knoten kann dabei natürlich auch voll werden, was zu einem weiteren Split führt. Dieser wird auf die gleiche Art behandelt wie ein Split eines Blattknotens und im Extremfall kann sich dieser Vorgang bis zur Wurzel fortsetzen. Wenn kein übergeordneter Knoten existiert (Überlauf des Wurzelknotens), wird einer erstellt. Der Baum wächst an der Wurzel! Es gibt keine andere Operation, die die Höhe des Baumes vergrößert. Die Überlaufbehandlung (Split) durchläuft den Baum vom Blatt zur Wurzel und die Anzahl der durchzuführenden Schritte ist im schlechtesten Fall (wie das Suchen der Einfügeposition) von der Höhe des Baumes abhängig. Alle Wege von der Wurzel zu einem Blatt bleiben gleich lang und alle Knoten (außer der Wurzel) sind immer mit mindestens *N* und maximal 2**N* Einträgen gefüllt.

Löschen

Beim Löschen wird ebenfalls auf Balance geachtet. Zunächst wird nach dem Blattknoten des zu löschenden Eintrags gesucht (Löschposition). Ist der Blattknoten zu mehr als *N* gefüllt, kann problemlos ein Eintrag entfernt werden. Wenn der Blattknoten genau *N* Einträge enthält, würde er weniger als halb voll und es wird ein *Ausgleich* mit einem seiner Nachbarknoten durchgeführt. Ist der Nachbarknoten auch nur mehr halb voll, so werden beide *verschmolzen* und damit zu einem fast vollständig gefüllten (2**N*-1). Dabei wird auch der Sortiereintrag für den entfernten Knoten aus dem übergeordneten Knoten entfernt, der wiederum weniger als halb voll werden könnte, was zu einem Ausgleich oder Verschmelzen auf der Ebene des übergeordneten Knotens führt. Das kann sich bis zum Wurzelknoten fortsetzen. Wenn der Wurzelknoten leer wird, so wird er entfernt und der Baum schrumpft an der Wurzel. Es gibt keine andere Operation, die die Höhe des Baumes verkleinert. Alle Wege von der Wurzel zu einem Blatt bleiben gleich lang und alle Knoten (außer der Wurzel) sind immer mit mindestens *N* und maximal 2**N* Einträgen gefüllt.

Einfüge- und Löschoperationen laufen von der Wurzel zum Blatt und, im schlechtesten Fall, wieder zurück zur Wurzel. Es sind somit bei einem B-Baum der Ordnung *N* maximal $2*\text{Log}_N|\text{Relation}|$ Schritte erforderlich. Man sagt, der Aufwand ist *logarithmisch*. Eine *ver-N-fachung* der Tupelanzahl führt zu einem Baum, der um *eine* Ebene höher ist.

Heap

Tabellen können im einfachsten Fall als so genannter *Heap* verwaltet werden. Jedes Tupel ist ein zusammengehöriger Speicherbereich mit einer festen persistenten Adresse (z.B. Position innerhalb einer Datei). Die Adresse eines Tupels ändert sich nicht, wenn andere Tupel eingefügt oder gelöscht werden. Durch Löschoperationen können Löcher im Heap entstehen, die für neu eingefügte Tupel, sofern sie darin Platz haben, verwendet werden. Ist kein Platz frei, wird an das Ende des Heaps angefügt, die Datenbank wächst dadurch. Das RDBMS verwaltet die freien Bereiche des Heaps und führt im Falle von großen freien Bereichen periodisch oder auf expliziten Wunsch des Administrators eine Reorganisation durch.

Im Fall eines Heaps kann in einem Index-Blattknoten die Tupelposition (Adresse) zur Referenzierung verwendet werden. Der Zugriff auf ein bestimmtes Tupel via Index besteht also aus dem Suchen im B-Baum des Index und dem anschließenden Lesen des referenzierten Tupels gemäß seiner Adresse.

Bei einer allfälligen Reorganisation eines Heaps (verdichten und damit verschieben der Tupel) müssen natürlich auch alle betroffenen Referenzen in Index-Knoten nachgeführt werden.

Clustered Index

Eine andere Möglichkeit besteht in der Verwendung eines so genannten *Clustered Index*. Dabei speichert man die Tupel direkt in den Blattknoten eines B^+-Baum-Index, der dadurch zu einem Clustered Index wird. Die Tupel können, wenn man Redundanz vermeiden will (und das will man normalerweise), nur in *einem* Index gespeichert sein, daher kann es nur *einen* Clustered Index geben. Alle anderen Index-Strukturen verwenden Referenzen auf die Tupel in den Blättern des Clustered Index. Dabei ergibt sich aber das Problem, dass diese nun keine festen Adressen mehr aufweisen, weil durch Einfüge- und Löschoperationen Blattknoten umorganisiert werden können. Ein unveränderliches Merkmal eines Tupels ist in diesem Fall aber der Schlüsselwert, der für die Sortierung im Clustered Index verwendet wurde. Dieser Schlüsselwert wird als eindeutige Referenz auf das Tupel verwendet. Falls der Clustered Index nicht UNIQUE ist, wird noch Zusatzinformation gespeichert, um ihn eindeutig zu machen.

Der Zugriff auf ein Tupel via Clustered Index erfolgt durch Suche im B-Baum bis zum zuständigen Blattknoten, der den Tupelwert enthält. Es ist keine weitere Indirektion erforderlich. Besonders effizient ist ein Clustered Index für Bereichsabfragen gemäß dem Index-Sortierkriterium, weil Tupel im gleichen Bereich in gemeinsamen Blattknoten abgespeichert sind. Sie sind also gebündelt (clustered). Daher ist in diesem Fall nur eine minimale Anzahl von Leseoperationen erforderlich.

Ein Zugriff via Non-Clustered Index ist dafür doppelt so aufwändig wie ohne. Neben der obligaten Suche im B-Baum des Indexes muss mit der gefundenen Tupelreferenz (das ist nun der Suchbegriff im Clustered Index) im Clustered Index ein zweites Mal gesucht werden. Es ist also sorgfältig abzuwägen, ob ein Clustered Index in Summe eine Ersparnis bringt oder nicht. Das RDBMS überlässt diese Entscheidung dem Programmierer, der über geeignete Optionen (z.B. CREATE UNIQUE CLUSTERED INDEX ...) ausdrückt, was er möchte.

Statistiken

Damit der Query-Compiler die Verwendung von Indices bestmöglich berücksichtigen kann, ist es notwendig, Wissen über die Treffsicherheit von Suchbegriffen zu besitzen. Wenn zum Beispiel mehrere Indices verfügbar sind, dann sollte jener verwendet werden, der die Ergebnismenge voraussichtlich am stärksten einschränkt. Ein Index auf einer Adresstabelle mit dem Attribut Postleitzahl sollte beispielsweise nur dann verwendet werden, wenn nicht alle Adressen aus dem gleichen Postleitzahlbereich stammen. Dann wäre es besser, gar keinen Index zu verwenden, oder einen Index auf einem anderen Attribut, etwa der Straße, usw.

Generell ist dazu statistisches Wissen über die Treffsicherheit von Suchbegriffen in Indices erforderlich. Die meisten RDBMS führen deshalb zu jedem Index statistische Zusatzinformationen mit (manchmal auch optional), die entweder automatisch aktualisiert oder durch ein eigenes Kommando berechnet werden. Die Syntax der benötigten Befehle ist RDBMS-abhängig. Die Statistiken werden meist aus einer kleinen Teilmenge der Tupel berechnet (Stichproben).

Vereinfachte Syntax DB2:

§ UpdateStatisticsStat = "RUNSTATS" "ON" "TABLE" Tablename.

Vereinfachte Syntax MS SQL Server:

§ UpdateStatisticsStat = "UPDATE" "STATISTICS" Tablename.

Elementare Bearbeitungsschritte

Der Ausführungsplan, der vom Compiler erzeugt wird, besteht aus einer RDBMS abhängigen Menge von Operationen, die aber in gewisse Gruppen eingeteilt werden können. Elementare Techniken finden sich bei allen RDBMS in der einen oder anderen Form wieder. Im Folgenden werden einige dieser Operationen beschrieben.

Table Scan. Dabei wird eine Tabelle vollständig und sequentiell (in physischer Reihenfolge) gelesen und eine Bedingung pro Tupel geprüft.

Index Scan. Dabei wird ein Index vollständig und in physischer Reihenfolge gelesen und eine Bedingung geprüft, die nur aus den indizierten Attributen besteht. Ein Index ist kompakter als die zu Grunde liegende Tabelle und kann schneller sequentiell gelesen werden (bei einem Clustered Index fallen die beiden Begriffe Table und Index natürlich zusammen). Wenn nur Attribute aus dem Index benötigt werden (Projektion), ist auch bei Verwendung eines Non-Clustered Index kein Zugriff auf die Tabellendaten erforderlich.

Index Seek. Dabei wird der (Non-Clustered) Index gemäß Sortierkriterium durchsucht. Das Ergebnis ist eine Teilmenge aus dem Index in einem Bereich (von, bis).

Nested Loop Join. Wir haben bereits die allgemeinste Art des Joins kennengelernt, den Nested Loop Join. Dabei wird in einer äußeren Schleife eine Tabelle durchlaufen und in einer inneren, eine andere. Der innere Schleifenrumpf wird verwendet, um die Join-Bedingung zu prüfen. Diese Join-Art wird verwendet, wenn kein Equi-Join vorliegt oder wenn die beteiligten Tabellen sehr klein sind.

Key Join. Eine effizientere Join-Art ist der so genannte Key-Join, den wir im Beispiel auch schon gesehen haben. Dabei wird mittels eines Index (falls verfügbar) das Durchlaufen der inneren Schleife stark eingeschränkt. Diese Join-Art kann nur für Equi-Joins eingesetzt werden.

Merge Join. Diese Join-Art wird ebenfalls für Equi-Joins verwendet, setzt aber keinen Index voraus. Beide Tabellen werden gemäß dem Equi-Kriterium sortiert und durch Zusammenmischen der zusammengehörigen Tupel in einem sequentiellen verzahnten Durchlauf durch beide Tabellen erzeugt. Falls ein geeigneter Index vorliegt kann das Sortieren einer oder beider Relationen entfallen wodurch der Merge Join sehr schnell wird.

Hash Join. Anwendungsfall ähnlich wie Merge-Join. Bei nicht allzu großen Tabellen kann für die innere Tabelle eine Hash-Tabelle im Hauptspeicher aufgebaut werden (linearer Aufwand). Diese Hash-Tabelle erlaubt die Abbildung eines Schlüssels (EQUI-Kriterium) auf ein Tupel in praktisch konstanter Zeit. Das Sortieren der beiden Tabellen kann dadurch entfallen. Als innere Tabelle wird die kleinere Tabelle verwendet, damit die Hash-Tabelle im Hauptspeicher Platz hat. Der Aufwand für den Join zweier Relationen mit N bzw. M Tupel ist in diesem Fall N + M.