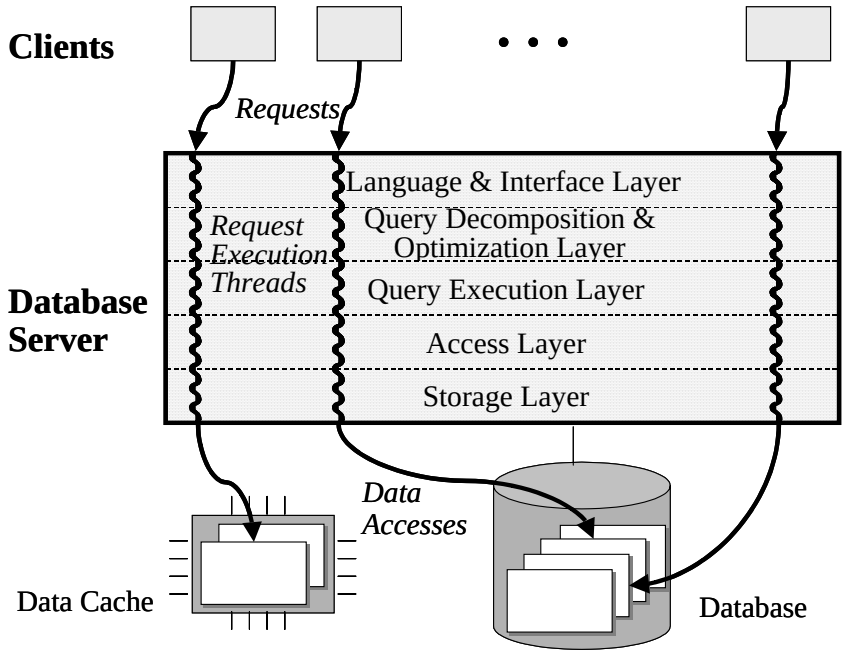
Kapitel 14: Datenspeicherung, Indexstrukturen und Anfrageauswertung

Praktisch alle Datenbanksysteme haben - zumindest in idealisierter Betrachtung - eine interne Schichtenarchitektur. Jede Schicht bietet an ihrer Schnittstelle bestimmte Methoden und Dienste an, mit deren Hilfe die Schnittstelle der nächsthöheren Schicht implementiert ist. Alle hier gemeinten Schichten liegen im selben Server; für Client-Programme ist nur die Schnittstelle der obersten Schicht sichtbar. Wenn ein Client-Auftrag (z.B. ein SQL-Kommando eines ESQL-Programms oder eines Web-Servlets) beim Server eintrifft, erzeugt oder verwendet er einen eigenen Thread, innerhalb dessen der Auftrag von Schicht zu Schicht in mehrere, primitivere Operationen zerlegt wird, bis er schließlich eine Reihe von Plattenzugriffen oder Seitenzugriffen auf einem hauptspeicherresidenten Daten-Cache erzeugt. Alle Threads zusammen bilden in der Regel einen gemeinsamen virtuellen Adressraum, in dem ein oder mehrere Betriebssystemprozesse auf einer CPU oder einem Parallelrechner (typischerweise einem SMP = Symmetric Multiprocessor) ablaufen.



Auf der ***Sprach- und Schnittstellenebene (language and interface layer)*** werden APIs für den Aufruf von Datenbankoperationen wie SQL-Kommandos bereitgestellt, also z.B. JDBC, ODBC oder auch produktspezifische Schnittstellen wie OCI (siehe auch Kapitel 6). Diese Kommandos werden zerlegt in interne Operatorbäume, die auf der nächsttieferen Ebene, der ***Anfragezerlegungs- und -optimierungsschicht (query decomposition and optimization layer)*** behandelt werden. Diese Operatorbäume entsprechen in erster Näherung relationenalgebraischen Ausdrücken, die aufgrund von Äquivalenzregeln sowie Kostenschätzungen für verschiedene Ausführungsplanvarianten optimiert werden. Diese Optimierung erfolgt möglichst zur Compile-Zeit der Anwendung, so daß einmal gene-

rierte Ausführungspläne bei wiederholter Ausführung der SQL-Anweisung bzw. des gesamten Anwendungsprogramms nicht jedes Mal erneut optimiert werden müssen.

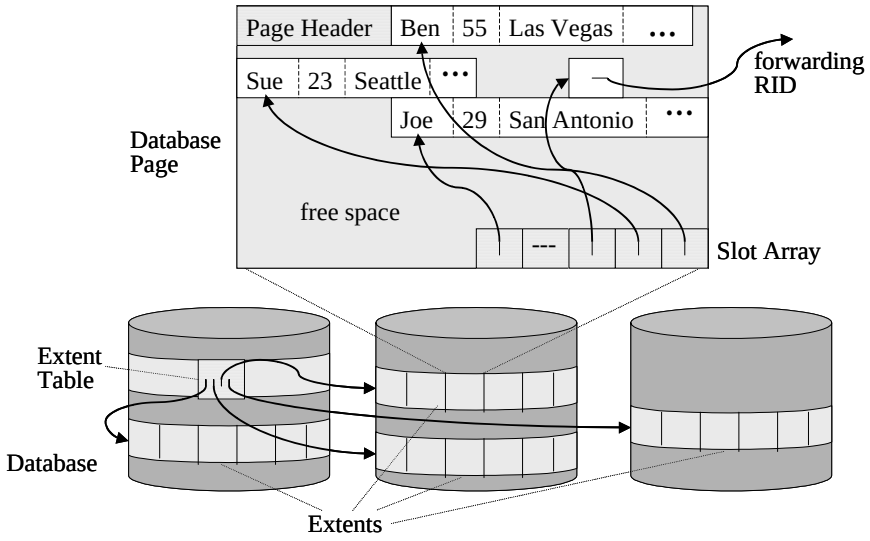
Jeder Operator in einem als Operatorbaum vorliegenden, optimierten Ausführungsplan wird auf einen bestimmtes Servermodul mit entsprechenden Algorithmen (z.B. zur Berechnung eines Equijoins) abgebildet. Diese Algorithmen und die Koordination des Kontroll- und Datenflusses zwischen den Operatoren wird von der ***Anfrageausführungsschicht (query execution layer)*** übernommen. Die einzelnen Operatoren wiederum erzeugen während ihrer Ausführung Record-Zugriffe sowie - zur Beschleunigung der Anfrage - Zugriffe auf sogenannte Indexstrukturen. Records und Indexstrukturen, letztere typischerweise als magnetplattenresidente Suchbäume implementiert, werden von der ***Zugriffsschicht (access layer)*** verwaltet. Alle Zugriffs- und Speicherungsstrukturen sind schlußendlich in Seiten (Blöcken) fester Größe (z.B. 32 KBytes) abgelegt, die die Transporteinheiten zwischen Magnetplatten und Hauptspeicher, genauer dem Daten-Cache, bilden. Im Daten-Cache werden jeweils die in jüngster Vergangenheit am häufigsten benutzten bzw. wichtigsten Seiten zum schnelleren Zugriff gepuffert; ein sogenannter Seitenersetzungsalgorithmus wie z.B. LRU (least recently used) steuert dynamisch den aktuellen Cache-Inhalt. Für Caching und Plattenzugriffe ist die ***Speicherungsschicht (storage layer)*** zuständig.

14.1 Wie Daten gespeichert werden

Alle Daten werden intern in Form von Records (Datensätzen) gespeichert, die aus einer Reihe von Feldern (Fields) bestehen und zusammen einen - in der Regel physisch zusammenhängenden - Bytestring variabler Länge bilden. Records werden in Seiten (Blöcken) fester Länge (z.B. 32 KBytes) abgelegt, die als Transportcontainer zwischen Magnetplatten und Daten-Cache im Hauptspeicher dienen. Seiten sind innerhalb eines Tablespaces, der als plattenresidenter virtueller Adressraum für eine oder mehrere Tabellen einer (relationalen) Datenbank dient, fortlaufend nummeriert. Physisch besteht ein Tablespace aus vorab allozierten Blöcken auf einer oder mehreren Magnetplatten, wobei typischerweise eine kleine Anzahl von größeren, physisch zusammenhängenden Bereichen, sog. Extents, verwendet wird. Mittels einer Extent-Tabelle kann die "logische" Seitennummer in eine "phyische" Plattenblockadresse umgerechnet werden.

Jede Seite hat einen Seiten-Header fester Größe mit Freispeicherverwaltungsinformationen u.ä. sowie einen variabel langen Seiten-Trailer, der ein - häufig "Slot-Array" - genanntes Pointer-Array mit den Byte-Offsets der in der Seite abgelegten Records enthält. Die Records selbst liegen zwischen Header und Trailer an beliebigen Offsets; wegen dynamischer Erweiterungen und Verschiebungen von Records können zwischen Records auch Lücken mit freiem Speicher liegen. ein Record wird adressiert durch eine sogenannte Row-ID bzw. Record-ID, kurz RID (seltener auch TID = Tupel-ID genannt), die aus der Tablespace-Nummer, der Seitennummer und der Slot-Nummer im Seiten-Trailer besteht. Durch die Indirektion über das Slot-Array können Records innerhalb einer Seite beliebig verschoben werden, ohne daß sich ihre - in Indexstrukturen als Zeiger verwendeten - Adressen ändern. Gelegentlich kann es vorkommen, daß ein Record aus Platzmangel auf eine andere Seite verlagert werden muß; in diesem Fall wird auf der ursprünglichen Seite ein kurzer "Forwarding-Eintrag" angelegt, so daß die RID auch dann unverändert bleibt. Bei gelegentlichen Reorganisationen, die vom Datenbankadministrator während ruhigerer Betriebszeiten gestartet werden können, werden die Forwarding-Einträge eliminiert und Zeiger in Indexstrukturen angepasst.

Ein Beispiel für die beschriebenen Speicherungsstrukturen - mit Personen-Records, die Felder wie Name (name), Alter (age), Stadt (city), etc. haben - zeigt die folgende Abbildung.



14.2 Wie auf Daten effizient zugegriffen wird (Indexstrukturen)

Zur Beschleunigung von simplen Selektionen, als Operatoren innerhalb eines Ausführungsplans, können vom Datenbankadministrator mit dem SQL-Kommando

CREATE [UNIQUE] INDEX index-name ON table (column, {, column ...})

Indexstrukturen angelegt werden. Ein Index auf Attributen A1, A2, ..., Ak erlaubt die effiziente Bestimmung aller Treffer für

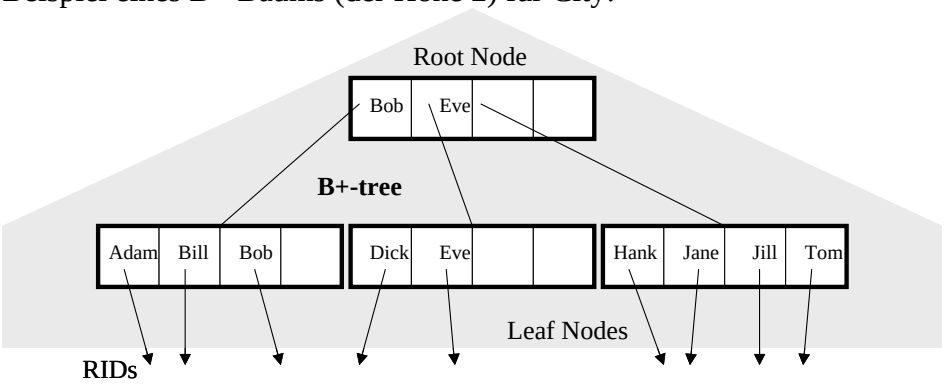
- Exact-Match-Selektionen der Form A1=wert1 $\wedge$ A2=wert2 $\wedge$ ... $\wedge$ Ak=wertk
  (z.B. City = 'Miami' (k=1) oder City = 'Paris' $\wedge$ State = 'Texas' (k=2)),
- Bereichs-Selektionen (Range Scans) der Form u1 $\leq$ A1 $\leq$ o1 $\wedge$ ... $\wedge$ uk $\leq$ Ak $\leq$ ok
  (z.B. 21 $\leq$ Age $\leq$ 30 (k=1) oder 21 $\leq$ Age $\leq$ 30 $\wedge$ Salaray $\geq$ 100 000 (k=2)) sowie
- Präfix-Match-Selektionen (mit einem Präfix der im Index verwalteten Felder)
  der Form u1 $\leq$ A1 $\leq$ o1 $\wedge$ u2 $\leq$ A2 $\leq$ o2 $\wedge$ ... $\wedge$ uj $\leq$ Aj $\leq$ oj (mit j < k)
  (z.B. 21 $\leq$ Age $\leq$ 30 bei einem Index über Age, Salary).

Indexstrukturen beschleunigen Anfragen, sofern die Anfrageoptimierung den Index entsprechend nutzt, belegen aber zusätzlichen Speicherplatz und bedeuten vor allem Mehraufwand für Änderungsoperationen. Wegen dieses Zielkonflikts obliegt es dem Datenbankadministrator, Indexstrukturen auf sinnvollen Feldkombinationen anzulegen; nur auf Primär- und Fremdschlüssel legen Datenbanksystem automatisch Indexstrukturen an. Bei dieser schwierigen Tuning-Aufgabe wird der Datenbankadministrator bei besseren Produkten von Optimierungswerkzeugen (z.B. sog. Index Wizards) unterstützt. Die Indexauswahl - als Teil der umfassenderen Aufgabe des sog. physischen Datenbankentwurfs - bleibt dennoch eine diffizile Fragestellung.

Ohne Index müssen Selektionen der o.a. Arten durch sequentiellen Scan ausgewertet werden, hätten also einen Berechnungsaufwand - gemessen in der Anzahl zu lesender Seiten der Datenbank -, der linear in der zugrundeliegenden Relationsgröße ist (in Kurznotation: O(n/C) mit n = Anzahl Records der Relation, C = Records pro Seite). Mit einem geeigneten Index verringert sich dieser Aufwand auf eine logarithmische Zeit in der Relationsgröße (in Kurnotation: $O(\log_k n/C)$ mit einer Basis k in der Größenordnung von einigen Hundert, s.u.). Um dies zu erreichen, sind Indexstrukturen intern als plattenresidente Suchbäume, genauer sog. B*-Bäume, implementiert.

Ein B*-Baum ist ein balancierter, hohler Mehrwegebaum für effizientes Suchen, Einfügen und Löschen von (Such-)Schlüsseln (und den damit verbundenen Records) auf seitenorientiertem Sekundärspeicher. Er beinhaltet auf Blattniveau Schlüssel-RID-Paare bzw. Schlüssel mit einer RID-Liste, die aus den RIDs derjenigen Records besteht, die den Schlüssel enthalten. Nichtblattknoten enthalten - im Schnitt - k geordnete Schlüssel und k+1 Sohnzeiger, wobei die Schlüssel in den Nichtblattknoten den Charakter von Wegweisern haben, um von der Wurzel ausgehend das Blatt zu finden, das den gesuchten Schlüssel enthält (oder enthalten müßte, wenn der Schlüssel überhaupt vorkäme). Der Baum ist jederzeit perfekt balanciert; alle Blätter haben also dieselbe Distanz zur Wurzel, nämlich $\text{ceil}(\log_k n/C) + 1$, woraus sich die logarithmische Suchzeit ergibt. k wird als der (mittlere) Fanout (Verzweigungsgrad) der Nichtblattknoten bezeichnet.

Beispiel eines B*-Baums (der Höhe 2) für City:



Formale Definition:
Ein Mehrwegebaum heißt B*-Baum der Ordnung (m,m*), $m \geq 1$, $m^* \geq 1$, wenn gilt:

- Jeder Nichtblattknoten außer der Wurzel enthält mindestens m und höchstens 2m Schlüssel.
- Ein Nichtblattknoten mit k Schlüssel x1, ..., xk hat genau k+1 Söhne t1, ..., t(k+1), so daß
  - für alle Schlüssel s im Teilbaum ti, $2 \leq i \leq k$, gilt $x(i-1) < s \leq xi$, und
  - für alle Schlüssel s im Teilbaum t1 gilt $s \leq x1$, und
  - für alle Schlüssel s im Teilbaum t(k+1) gilt $xk < s$.
- Alle Blätter haben dasselbe Niveau (d.h. Distanz von der Wurzel).
- Jedes Blatt enthält mindestens m* und höchstens 2m* Schlüssel
  bzw. Schlüssel-RID(-Listen)-Paare.

Die entscheidende Invariante eines B*-Baums besteht also darin, daß durch die Wegweiser eines Knotens der Suchbereich in disjunkte Teilbereiche partitioniert wird. Dies ermöglicht einen Divide-

and-Conquer-Ansatz für das effiziente Suchen. Die Suche nach einem Schlüssel s beginnt bei der Wurzel und arbeitet sich top-down bis zu einem Blatt herunter. Auf jeder Baumstufe wird ein Knoten nach dem kleinsten Schlüssel xi durchsucht, für den $s \leq xi$ gilt; dies erfolgt mittels binärer Suche auf den innerhalb eines Knotens sortiert abgelegten Wegweisern. Anschließend wird die Suche im Teilbaum ti fortgesetzt (bzw. t(k+1) falls es kein xi gibt mit $s \leq xi$), bis schlußendlich ein Blatt erreicht ist. Dort endet die Suche entweder erfolgreich - und liefert dann die zum gesuchten Schlüssel gehörenden RIDs zurück - oder erfolglos - mit einem Returncode, der anzeigt, daß der gesuchte Schlüssel überhaupt nicht vorhanden ist. Im o.a. B*-Baum etwa würde die Suche nach 'Dick' dem Wegweiser links vom Eintrag 'Eve' in der Wurzel folgen und dann im darunter hängenden Blatt erfolgreich enden. Bereichsanfragen suchen den kleinsten Schlüssel des spezifizierten Bereichs, also den "linken Rand", und traversieren dann alle Blätter, bis ein Schlüssel erreicht wird, der größer ist als der "rechte Rand" des Suchbereichs. Zum Zwecke dieser Traveriserung sind benachbarte Blätter durch Zeiger verkettet.

Pseudocode für Suchen von Schlüssel s in B*-Baum mit Wurzel t:

```
t habe k Schlüssel x1, ..., xk und k+1 Söhne t1, ..., t(k+1)
(letzteres sofern t kein Blatt ist)
Bestimme den kleinsten Schlüssel xi, so daß s ≤ xi
if s = xi (für ein i ≤ k) und t ist ein Blatt
then Schlüssel gefunden
else
    if t ist kein Blatt then
        if s ≤ xi (für ein i ≤ k)
        then suche s im Teilbaum ti
        else suche s im Teilbaum t(k+1) fi
    else Schlüssel s ist nicht vorhanden fi
 fi
```

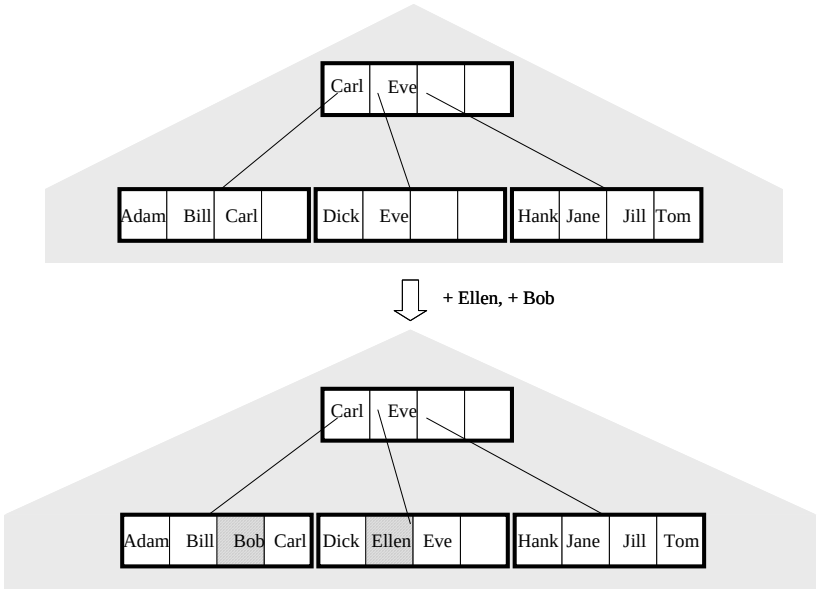
Statt einer festen Maximalanzahl 2m von Wegweisern in einem Knoten wird wegen der variablen Länge der Schlüssel eine variable Anzahl von Wegweisern gespeichert. Da die Einträge in den Nichtblattknoten nur der richtigen Verzweigung beim Suchen dienen, müssen diese nicht die vollständigen Schlüssel wiedergeben, sondern können auf geeignete Präfixe reduziert werden. Zum Beispiel könnte man im o.a. B*-Baum den Wegweiser 'Eve' in der Wurzel durch den kürzeren Separator 'F' ersetzen, ohne die Bauminvariante zu verletzen. Dadurch kann der Fanout des B*-Baums noch weiter erhöht werden, wodurch die Baumtiefe sinkt. Typische Fanouts in der Praxis liegen bei 100 bis 1000, so daß auch auf sehr großen Datenbeständen B*-Bäume selten höher als 3 sind. Dabei sind - bei sinnvoller Systemkonfiguration - alle Baumknoten außer den Blättern so gut wie cache-resident, so daß jede Exact-Match-Selektion mittels Index mit einem einzigen Plattenzugriff ausgeführt werden kann.

Beim Einfügen eines neuen Schlüssels wird zunächst das Blatt gesucht, in dem der Schlüssel abgelegt sein müßte, wenn er schon vorher vorhanden gewesen wäre. Wenn in diesem Blatt ausreichend freier Platz ist, wird der neue Schlüssel dort gespeichert. Wenn kein Platz ist, erfolgt ein sog. Knoten-Split. Dazu wird eine neue Seite von der Freispeicherverwaltung angefordert, die zum rechten Nachbarn des Blattes wird. Der Inhalt des "übergelaufenen" Blatts wird gleichmäßig zwischen altem und neuem Blatt aufgeteilt. Anschließend wird ein neuer Wegweiser-Eintrag erstellt, der einen Separator zwischen altem und neuem Blatt (z.B. den größten im alten Blatt verbleibenden Schlüssel) so-

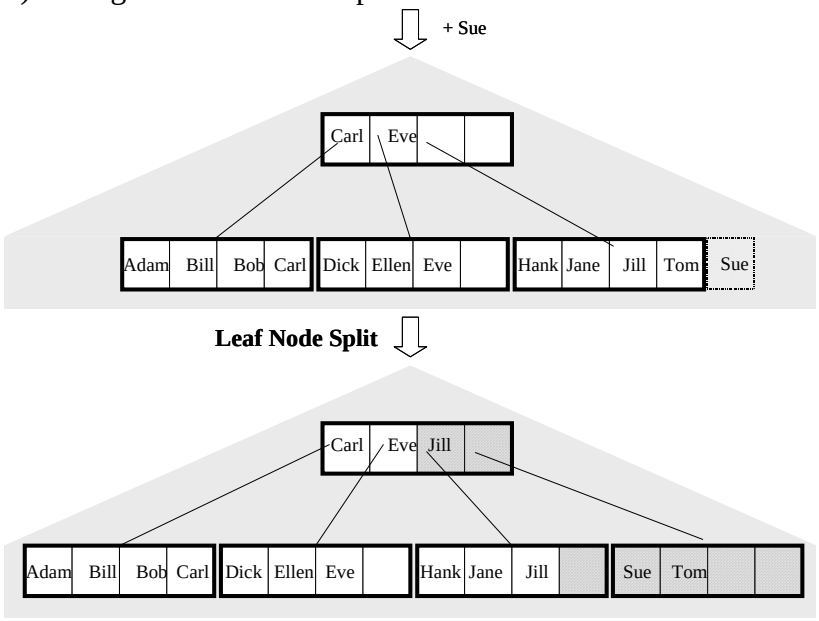
wie einen Zeiger auf das neue Blatt enthält. Dieser Wegweiser wird dann im Vater der beiden betroffenen Blätter eingefügt. Da es auch im Vater Platzknappheit geben kann, muß der Vaterknoten selbst evtl. analog geteilt werden, und ein Split kann sich rekursiv bis zur Wurzel fortpflanzen. Wenn die Wurzel geteilt werden muß, wird eine Wurzel erzeugt, so daß die Baumhöhe um eins wächst.

Die folgenden Abbildungen zeigen die verschiedenen Fälle beim Einfügen.

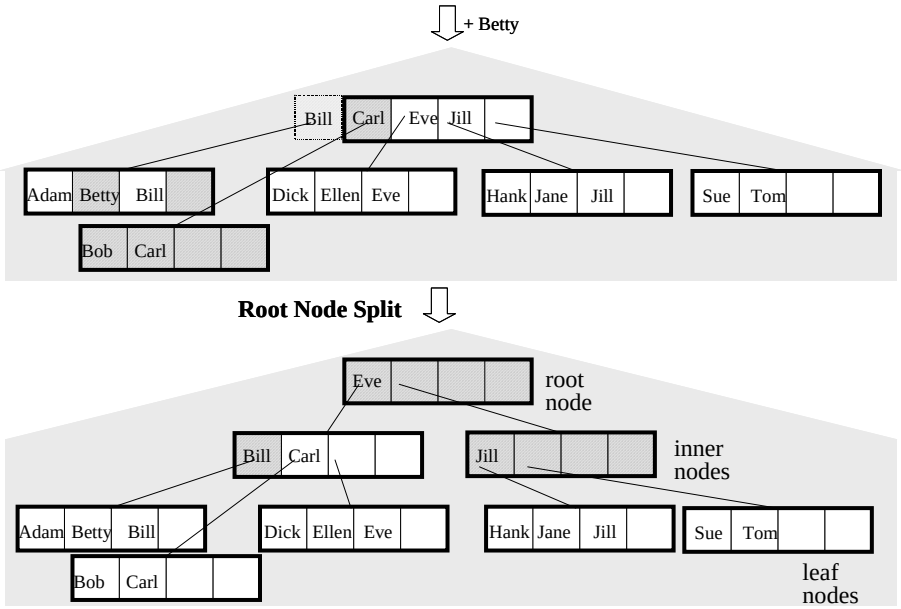
a) Einfügen von 'Ellen' und 'Bob' mit hinreichend freiem Platz im Blattknoten:



b) Einfügen von 'Sue' mit Split eines Blattknotens:



c) Einfügen von 'Betty' mit Blatt-Split und Wurzel-Split:



Pseudocode für das Einfügen eines neuen Schlüssels e in einen B*-Baum:

```
Suche nach einzufügendem Schlüssel e
if e ist noch nicht vorhanden then
  Sei t das Blatt, bei dem die Suche erfolglos geendet hat
  repeat
    if t hat weniger als 2m* bzw. 2m Schlüssel (d.h. ist nicht voll)
    then füge e in t ein
    else /* Knoten-Split */
      Bestimme Median s der 2m* +1 bzw. 2m+1 Schlüssel inkl. e
      Erzeuge Bruderknoten t‘ /* Grow-Phase */
      if t ist Blattknoten then
         Speichere Schlüssel ≤ s in t und Schlüssel > s in t‘
      else Speichere Schlüssel < s (mit Sohnzeigern) in t und
                        Schlüssel > s (mit Sohnzeigern) in t‘ fi
      if t ist Wurzel /* Post-Phase */
      then Erzeuge neue Wurzel r mit Schlüssel s und Zeigern auf t und t‘
      else Betrachte Vater von t als neues t und s (mit Zeiger auf t‘) als e fi
    fi
  until kein Knoten-Split mehr erfolgt
fi
```

14.3 Wie Anfragen ausgeführt werden

Anfragen werden übersetzt in Operatorbäume, die im wesentlichen relationenalgebraischen Ausdrücken auf Mengen, Multimengen und Listen. Gegenüber der Relationenalgebra aus Kapitel 2 gibt es weitere Operatoren, die sich an den Speicherungs- und Zugriffsstrukturen orientieren, z.B. einen Table-Scan-Operator zum sequentiellen Lesen aller Records einer Tabelle, einen Index-Scan-Operator zum Bestimmen aller RIDs zu einem Suchschlüssel, usw. Die wichtigsten Operatoren einer solchen DBS-internen Algebra sind im folgenden kurz zusammengestellt. In den meisten kommerziellen Datenbanksystemen kann man sich den für eine Anfrage erzeugten Operatorbaum mittels des Explain-Kommandos anschauen.

Table-Scan (TABLE ACCESS FULL):
Sequentielles Lesen aller Tupel einer Relation,
wobei auf jedem Tupel ein "Single-Scan"-Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.

RID-Zugriff (TABLE ACCESS BY ROWID):
Direkter Zugriff auf alle Tupel einer Liste von RIDs,
wobei auf jedem Tupel ein "Single-Scan"-Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.
Die RID-Liste sollte nach Seitennummern sortiert sein; Seiten mit kleinem "Abstand" auf derselben Platte (z.B. im selben Zylinder) sollten mit einem einzigen Plattenzugriff
gelesen werden (multi-block I/O, list prefetch).

Index-Scan (INDEX RANGE SCAN oder INDEX UNIQUE SCAN):
Zugriff auf alle TIDs von Tupeln, die ein Index-Suchprädikat der Form $wert1 \leq KEY \leq wert2$ erfüllen, wobei KEY der Suchschlüssel des Index ist (ggf. eine Attributkombination).
Realisiert ist dies durch eine Indexsuche nach wert1 mit einem nachfolgenden "Blatt-Scan" bis zum Erreichen des ersten Schlüssels, der größer als wert2 ist.

Sortieren (SORT):
Sortieren einer Tupelmenge nach einem Attribut bzw. einer Attributkombination.
Dabei werden Sortieralgorithmen für Externspeicher verwendet (z.B. Mehrwegemischen).

Durchschnitt, Vereinigung, Differenz (INTERSECTION, UNION, MINUS):
Anwenden von Mengenoperationen auf Tupelmengen oder TID-Listen.
Dies wird in der Regel erst nach vorhergehendem Sortieren angewandt.

Selektion bzw. Filterung (FILTER):
Überprüfen einer Booleschen Bedingung auf den Tupeln einer Tupelliste

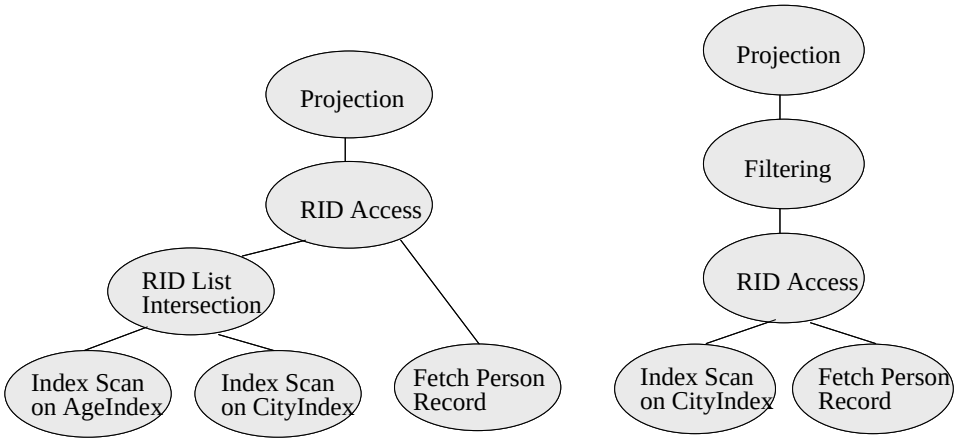
Projektion für Mengen (PROJ-SET):
Projektion von Attributen einer nach diesen Attributen sortierten Tupelliste mit Duplikateliminierung

Projektion für Multimengen (PROJ-MULTI):
Projektion von Attributen einer Tupelliste

Für eine SQL-Anfrage wie

```
Select Name, City, Zipcode, Street
From Person
Where Age < 30 And City = 'Austin'
```

erzeugt die Anfragezerlegungs- und -optimierungsschicht etwa einen der folgenden beiden Operatorbäume als Ausführungsplan:



In Oracle8i werden diese Ausführungspläne folgendermaßen dargestellt:

```
SELECT STATEMENT
        TABLE ACCESS            BY ROWID    Person
                INTERSECT
                        INDEX RANGE SCAN    AgeIndex
                        INDEX RANGE SCAN    CityIndex
```

bzw.

```
SELECT STATEMENT
        FILTER
                TABLE ACCESS    BY ROWID    Person
                        INDEX RANGE SCAN    CityIndex
```

Für Joins und Gruppierungen mit Aggregation gibt es weitere interne Operatoren als Implementierungsvarianten, beispielsweise die folgenden beiden Basisvarianten (für die es wiederum weitere optimierte Varianten gibt):

Nested-Loop-Join (NESTED LOOPS):
Berechnung eines Joins zwischen zwei Tupelmengen (mit Angabe einer expliziten Joinbedingung) mittels einer Doppelschleife.

Merge-Join (MERGE JOIN):
Berechnung eines Equi-Joins zwischen zwei nach dem Join-Attribut sortierten Tupellisten mittels Mischen der Listen.
Dieses Join-Berechnungs-Verfahren heißt häufig auch Sort-Merge-Join, weil u.U. vor dem Mischen noch Sortierschritte stattfinden müssen.

Hash-Join (HASH JOIN):
Berechnung eines Equi-Joins zwischen zwei Tupelmengen durch Abbilden beider Mengen auf eine Hash-Tabelle (mit einer für beide Tupelmengen identischen Hash-Funktion über dem Join-Attribut), so daß Tupel mit gleichem Attributwert auf denselben Eintrag (bzw. Bucket) der Hash-Tabelle abgebildet werden.

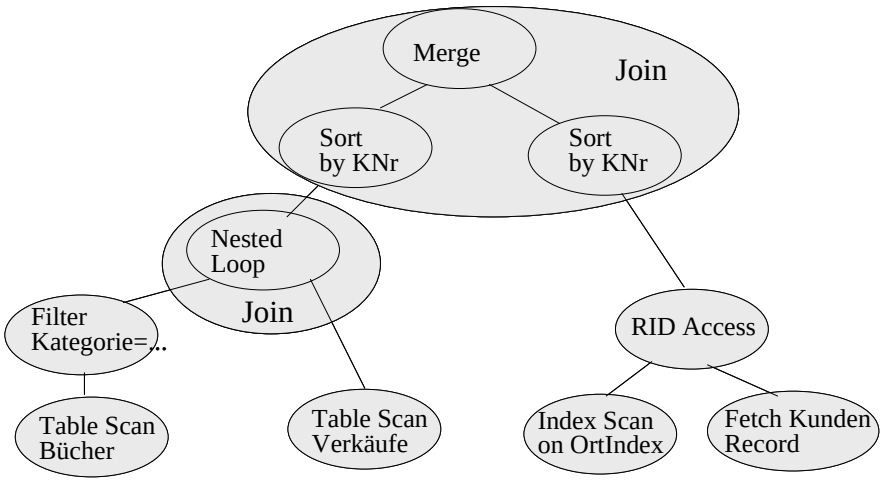
Gruppierung mit Aggregation (GROUP):
Berechnung eines Aggregationsfunktionswerts pro Gruppe auf einer nach dem Gruppierungsattribut sortierten Tupelliste durch sequentielles Lesen aller Tupel.

Hash-Gruppierung mit Aggregation (HASH GROUP):
Berechnung eines Aggegationsfunktionswerts pro Gruppe durch Abbilden aller Tupel auf eine Hashtabelle mit einer Hashfunktion auf dem Gruppierungsattribut und - falls erforderlich - anschließendem SORT und GROUP für jede Hashklasse.

Eine komplexere Query wie

```
Select KNr, Name
From Kunden K, Verkäufe V, Bücher B
Where K.Ort = 'Saarbrücken'
And K.KNr = V.KNr And V.ISBN = B.ISBN
And B.Kategorie = 'Kochbücher'
```

könnte beispielsweise zum folgenden Ausführungsplan führen:



In Oracle8i würde dieser Ausführungsplan folgendermaßen dargestellt:

```
SELECT STATEMENT
        MERGE JOIN
                SORT
                        NESTED LOOP
                                FILTER
                                        TABLE ACCESS    FULL        Bücher
                                TABLE ACCESS            FULL        Verkäufe
                SORT
                        TABLE ACCESS                    BY ROWID    Kunden
                                INDEX RANGE SCAN                    OrtIndex
```

Ein solcher Operatorbaum wird unter Anwendung relationenalgebraischer Äquivalenzregeln erzeugt, beispielsweise der Distributivität von Selektionen (Filter), die sich nur auf eine Tabelle beziehen, über Joins sowie der Assoziativität und Kommutativität von Equi-Joins. Der Anfrageoptimierer generiert eine (u.U. sehr große) Menge alternativer Ausführungspläne. Für jeden Kandidaten werden dann die Zugriffskosten, insbesondere die erwartete Anzahl der Plattenzugriffe, geschätzt, und der beste Ausführungsplan unter den Kandidaten wird schlußendlich gewählt. Diese Optimierung ist ein sehr schwieriges Problem, so daß reale Datenbanksysteme auf Heuristiken zurückgreifen. Bereits die Kostenschätzung für einen einzigen Plan ist schwierig, da die Ausführungskosten in starkem Maße von der Verteilung der Daten selbst abhängen (z.B. von Häufigkeiten von Attributwerten).

Die mengenorientierten Operatoren der Anfrageausführungsschicht eines Datenbanksystems sind sehr gut zur Parallelisierung geeignet, indem man einen Operator "klont" und jedem Klon eine Partition der Eingabedaten übergibt. Dabei kann sowohl I/O- als auch CPU-Parallelität mit sehr gutem Speedup ausgenutzt werden. Die kommerziell führenden Datenbanksysteme laufen alle mit sehr guter Leistung auf Parallelrechnern (mit u.U. 64 und mehr Prozessoren) und parallelen Plattensystemen (sog. Disk-Arrays, mit u.U. Hunderten von Platten).

Der Datenfluß zwischen einem Produzenten- und einem Konsumentenoperator im Operatorbaum (die in einer Kind-Vater-Beziehung im Baum stehen) kann auf zwei Arten realisiert werden. Zum einen kann jeder Operator sein (Teil-)Ergebnis materialisieren, indem er es komplett berechnet und im Hauptspeicher oder in einem temporären Arbeitsbereich auf Magnetplatte(n) ablegt, von wo es der nächste Operator liest. Alternativ dazu kann zwischen bestimmten Operatoren eine Pipeline im Sinne einer Fließbandverarbeitung aufgebaut werden, bei der ein Produzent so schnell wie möglich Tupel seines Output-Stroms, also bevor das gesamt (Teil-)Ergebnis berechnet ist, an den Konsumenten weitergibt. Pipelining vermeidet die Materialisierung von Zwischenergebnissen, ist jedoch deutlich schwieriger zu implementieren, insbesondere in Verbindung mit der Datenparallelität einzelner Operatoren.

Weiterführende Literatur zu Kapitel 14:

- T. Härder, E. Rahm: Datenbanksysteme - Konzepte und Techniken der Implementierung, Springer, 1999
- H. Garcia-Molina, J. Ullman, J. Widom: Database System Implementation, Prentice Hall, 1999
- G. Saake, A. Heuer: Datenbanken - Implementierungstechniken, MITP-Verlag, 1999
- C.T. Yu, W. Meng: Principles of Database Query Processing for Advanced Applications, Morgan Kaufmann, 1998