Kapitel 4

Physikalische Datenorganisation

4.1 Grundlagen

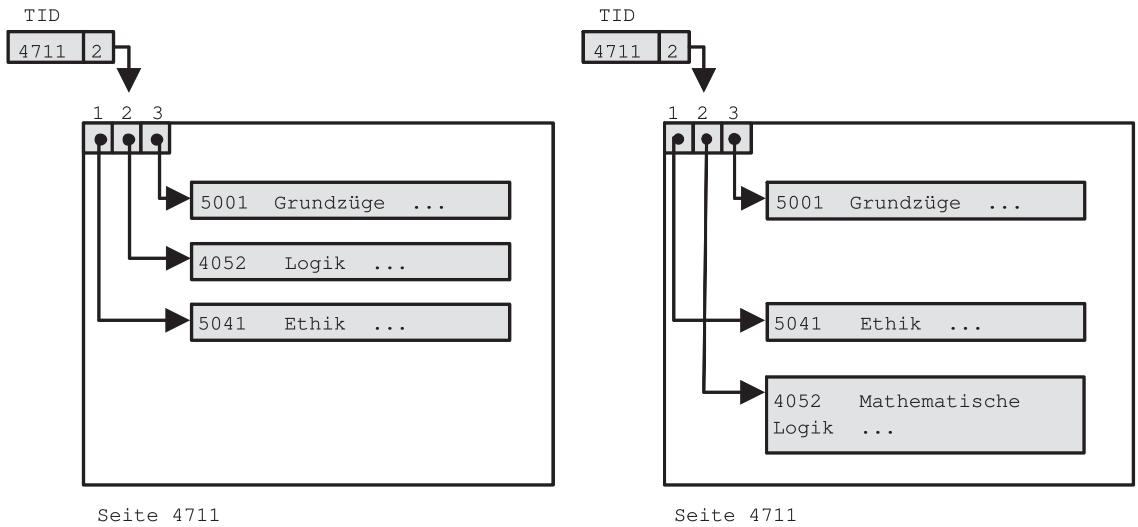

Abbildung 4.1: Verschieben eines Tupels innerhalb einer Seite

Die grundsätzliche Aufgabe bei der Realisierung eines internen Modells besteht aus dem Abspeichern von Datentupeln, genannt *Records*, in einem *File*. Jedes Record hat ein festes Record-Format und besteht aus mehreren Feldern meistens fester, manchmal auch variabler Länge mit zugeordnetem Datentyp. Folgende Operationen sind erforderlich:

- **INSERT:** Einfügen eines Records
- **DELETE:** Löschen eines Records
- **MODIFY:** Modifizieren eines Records
- **LOOKUP:** Suchen eines Records mit bestimmtem Wert in bestimmten Feldern.

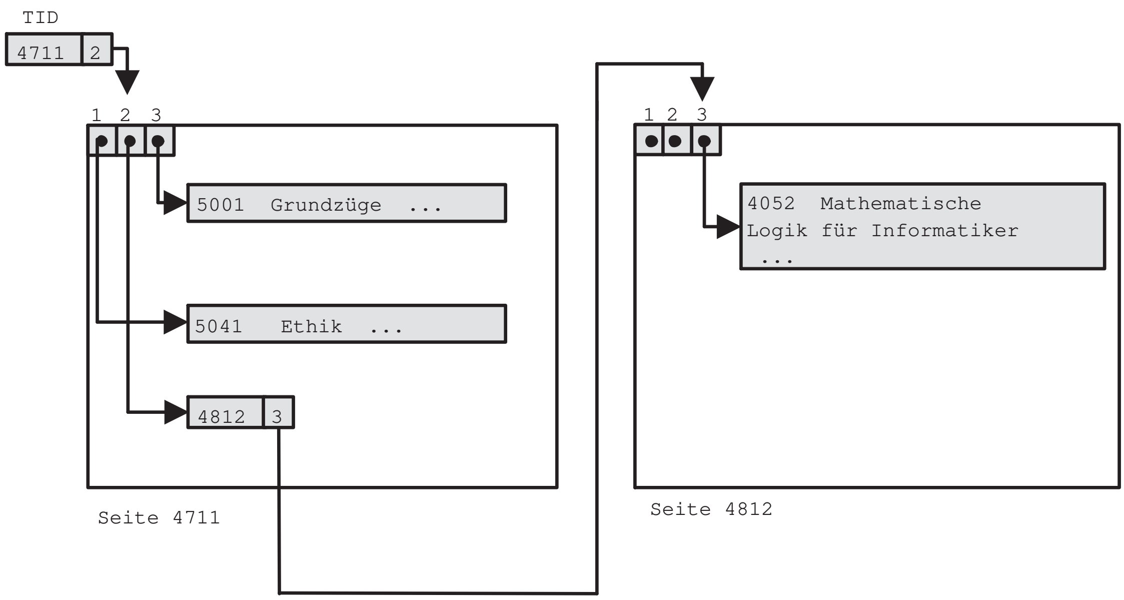

Abbildung 4.2: Verdrängen eines Tupels von einer Seite

Files werden abgelegt im Hintergrundspeicher (Magnetplatte), der aus *Blöcken* fester Größe (etwa 2^9 - 2^{12} Bytes) besteht, die direkt adressierbar sind. Ein File ist verteilt über mehrere Blöcke, ein Block enthält mehrere Records. Records werden nicht über Blockgrenzen verteilt. Einige Bytes des Blockes sind unbenutzt, einige werden für den *header* gebraucht, der Blockinformationen (Verzeigerung, Record-Interpretation) enthält.

Die *Adresse* eines Records besteht entweder aus der Blockadresse und einem *Offset* (Anzahl der Bytes vom Blockanfang bis zum Record) oder wird durch den sogenannten *Tupel-Identifikator* (TID) gegeben. Der Tupel-Identifikator besteht aus der Blockadresse und einer Nummer eines Eintrags in der internen Datensatztabelle, der auf das entsprechende Record verweist. Sofern genug Information bekannt ist, um ein Record im Block zu identifizieren, reicht auch die Blockadresse. Blockzeiger und Tupel-Identifikatoren erlauben das Verschieben der Records im Block (*unpinned records*), Record-Zeiger setzen fixierte Records voraus (*pinned records*), da durch Verschieben eines Records Verweise von außerhalb mißinterpretiert würden (*dangling pointers*).

Abbildung 4.1 zeigt das Verschieben eines Datentupels innerhalb seiner ursprünglichen Seite; in Abbildung 4.2 wird das Record schließlich auf eine weitere Seite verdrängt.

Das Lesen und Schreiben von Records kann nur im Hauptspeicher geschehen. Die Blockladezeit ist deutlich größer als die Zeit, die zum Durchsuchen des Blockes nach bestimmten Records gebraucht wird. Daher ist für Komplexitätsabschätzungen nur die Anzahl der Blockzugriffe relevant.

Zur Umsetzung des Entity-Relationship-Modells verwenden wir

- Records für Entities
- Records für Relationships (pro konkrete Beziehung ein Record mit TID-Tupel)

4.2 Heap-File

Die einfachste Methode zur Abspeicherung eines Files besteht darin, alle Records hintereinander zu schreiben. Die Operationen arbeiten wie folgt:

- **INSERT:** Record am Ende einfügen (ggf. überschriebene Records nutzen)
- **DELETE:** Lösch-Bit setzen
- **MODIFY:** Record überschreiben
- **LOOKUP:** Gesamtes File durchsuchen

Bei großen Files ist der lineare Aufwand für LOOKUP nicht mehr vertretbar. Gesucht ist daher eine Organisationsform, die

- ein effizientes LOOKUP erlaubt,
- die restlichen Operationen nicht ineffizient macht,
- wenig zusätzlichen Platz braucht.

4.3 Hashing

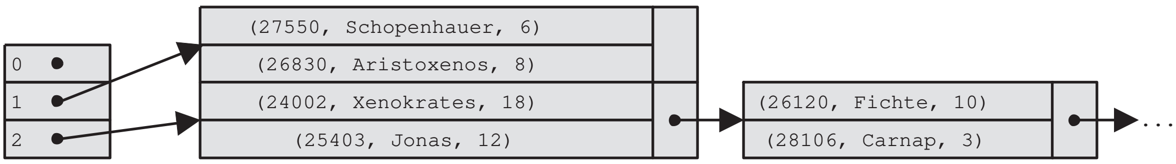

Abbildung 4.3: Hash-Tabelle mit Einstieg in Behälter

Die grundlegende Idee beim *offenen Hashing* ist es, die Records des Files auf mehrere Behälter (englisch: *Bucket*) aufzuteilen, die jeweils aus einer Folge von verzeigerten Blöcken bestehen. Es gibt eine *Hash-Funktion* h, die einen Schlüssel als Argument erhält und ihn auf die Bucket-Nummer abbildet, unter der der Block gespeichert ist, welcher das Record mit diesem Schlüssel enthält. Sei B die Menge der Buckets, sei V die Menge der möglichen Record-Schlüssel, dann gilt gewöhnlich $|V| \gg |B|$.

Beispiel für eine Hash-Funktion:

Fasse den Schlüssel v als k Gruppen von jeweils n Bits auf. Sei d_i die i-te Gruppe als natürliche Zahl interpretiert. Setze

$$h(v) = (\sum_{i=1}^{k} d_i) mod\, B$$

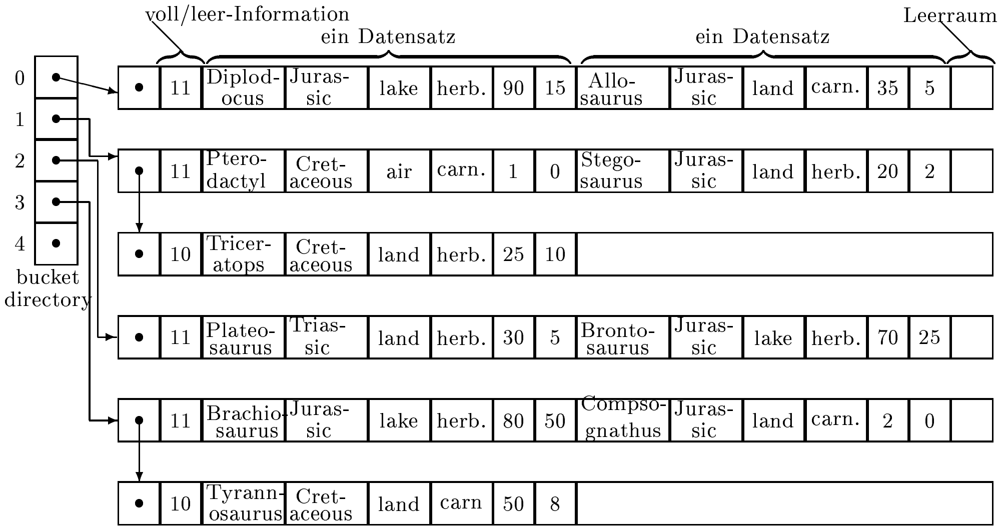

Abbildung 4.4: Hash-Organisation vor Einfügen von Elasmosaurus

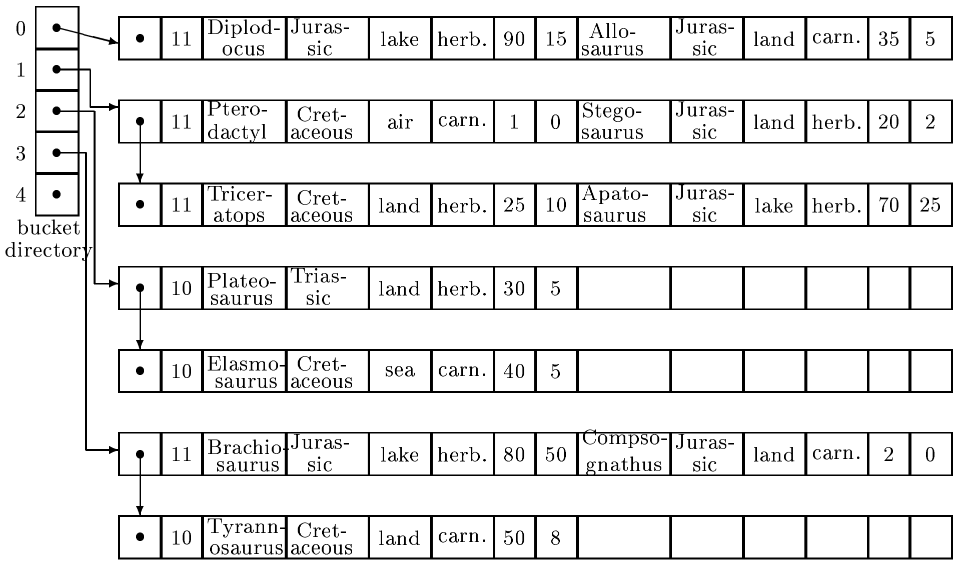

Abbildung 4.5: Hash-Organisation nach Einfügen von Elasmosaurus und Umbenennen

Im Bucket-Directory findet sich als $h(v)$-ter Eintrag der Verweis auf den Anfang einer Liste von Blöcken, unter denen das Record mit Schlüssel v zu finden ist. Abbildung 4.3 zeigt eine Hash-Tabelle, deren Hash-Funktion h die Personalnummer x durch $h(x) = x \bmod 3$ auf das Intervall [0..2] abbildet.

Falls B klein ist, kann sich das Bucket-Directory im Hauptspeicher befinden; andernfalls ist es über mehrere Blöcke im Hintergrundspeicher verteilt, von denen zunächst der zuständige Block geladen werden muß.

Jeder Block enthält neben dem Zeiger auf den Folgeblock noch jeweils 1 Bit pro Subblock (Platz für ein Record), welches angibt, ob dieser Subblock leer (also beschreibbar) oder gefüllt (also lesbar) ist. Soll die Möglichkeit von *dangling pointers* grundsätzlich ausgeschlossen werden, müßten gelöschte Records mit einem weiteren, dritten Zustand versehen werden, der dafür sorgt, daß dieser Speicherplatz bis zum generellen Aufräumen nicht wieder verwendet wird.

Zu einem Record mit Schlüssel v laufen die Operationen wie folgt ab:

- **LOOKUP:**
  Berechne $h(v) = i$. Lies den für i zuständigen Directory-Block ein, und beginne bei der für i vermerkten Startadresse mit dem Durchsuchen aller Blöcke.
- **MODIFY:**
  Falls Schlüssel von Änderung betroffen: DELETE und INSERT durchführen. Falls Schlüssel von Änderung nicht betroffen: LOOKUP durchführen und dann Überschreiben.
- **INSERT:**
  Zunächst LOOKUP durchführen. Falls Satz mit v vorhanden: Fehler. Sonst: Freien Platz im Block überschreiben und ggf. neuen Block anfordern.
- **DELETE:**
  Zunächst LOOKUP durchführen. Bei Record Löschbit setzen.

Der Aufwand aller Operationen hängt davon ab, wie gleichmäßig die Hash-Funktion ihre Funktionswerte auf die Buckets verteilt und wie viele Blöcke im Mittel ein Bucket enthält. Im günstigsten Fall ist nur ein Directory-Zugriff und ein Datenblock-Zugriff erforderlich und ggf. ein Blockzugriff beim Zurückschreiben. Im ungünstigsten Fall sind alle Records in dasselbe Bucket gehasht worden und daher müssen ggf. alle Blöcke durchlaufen werden.

Beispiel für offenes Hashing (übernommen aus *Ullman, Kapitel 2*):

Abbildungen 4.4 und 4.5 zeigen die Verwaltung von Dinosaurier-Records. Verwendet wird eine Hash-Funktion h, die einen Schlüssel v abbildet auf die Länge von $v \ mod \ 5$. Pro Block können zwei Records mit Angaben zum Dinosaurier gespeichert werden sowie im Header des Blocks zwei Bits zum Frei/Belegt-Status der Subblocks.

Abbildung 4.4 zeigt die Ausgangssituation. Nun werde `Elasmosaurus` (Hashwert = 2) eingefügt. Hierzu muß ein neuer Block für Bucket 2 angehängt werden. Dann werde `Brontosaurus` umgetauft in `Apatosaurus`. Da diese Änderung den Schlüssel berührt, muß das Record gelöscht und modifiziert neu eingetragen werden. Abbildung 4.5 zeigt das Ergebnis.

Bei geschickt gewählter Hash-Funktion werden sehr kurze Zugriffszeiten erreicht, sofern das Bucket-Directory der Zahl der benötigten Blöcke angepaßt ist. Bei statischem Datenbestand läßt sich dies leicht erreichen. Problematisch wird es bei dynamisch wachsendem Datenbestand. Um immer größer werdende Buckets zu vermeiden, muß von Zeit zu Zeit eine völlige Neuorganisation der Hash-Tabelle durchgeführt werden. Da dies sehr zeitaufwendig ist, wurde als Alternative das *erweiterbare Hashing* entwickelt.

4.4 Erweiterbares Hashing

Durch $h(x) = dp$ wird nun die binäre Darstellung des Funktionswerts der Hash-Funktion in zwei Teile zerlegt. Der vordere Teil d gibt die Position des zuständigen Behälters innerhalb des Hashing-Verzeichnis an. Die Größe von d wird die *globale Tiefe t* genannt. p ist der zur Zeit nicht benutzte Teil des Schlüssels.

	h(x)	
x	d	p
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001

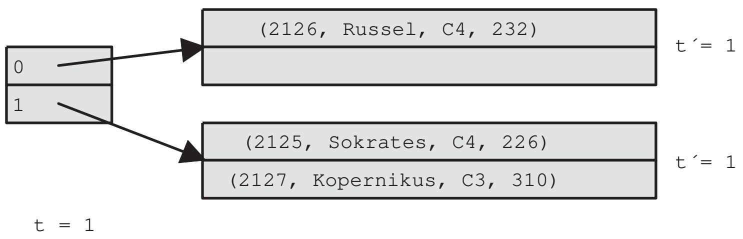

Abbildung 4.6: Hash-Index mit globaler Tiefe 1

	h(x)	
x	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001

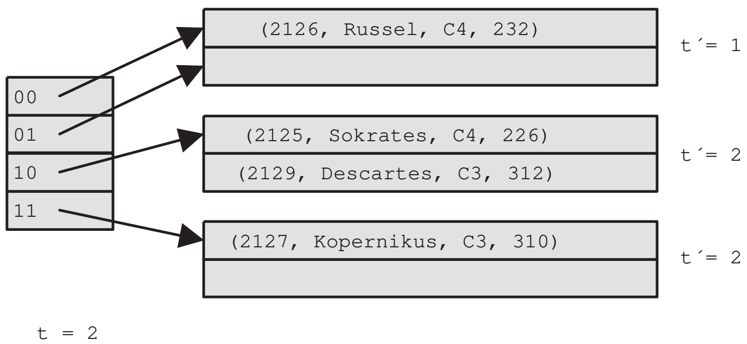

Abbildung 4.7: Hash-Index mit globaler Tiefe 2

Soll ein neuer Datensatz in einen bereits vollen Behälter eingetragen werden, erfolgt eine Aufteilung anhand eines weiteren Bit des bisher unbenutzen Teils p. Ist die globale Tiefe nicht ausreichend, um den Verweis auf den neuen Behälter eintragen zu können, muß das Verzeichnis verdoppelt werden.

Die *lokale Tiefe t'* eines Behälters gibt an, wieviele Bits des Schlüssels für diesen Behälter tatsächlich verwendet werden.

Abbildung 4.6 zeigt eine Hashing-Organisation für Datentupel aus dem Entity-Typ *Dozent* mit den Attributen *PersNr, Name, Rang, Raum*. Je zwei Records finden in einem Behälter Platz. Als Hash-Funktion wird die umgekehrte binäre Darstellung der Personalnummer verwendet. Die globale Tiefe beträgt zur Zeit 1, d.h. nur das vorderste Bit entscheidet über den Index des zuständigen Behälters.

Nun soll `Descartes` eingefügt werden. Das vorderste Bit seines Hash-Werts ist 1 und bildet ihn auf den bereits vollen, mit `Sokrates` und `Kopernikus` gefüllten Behälter ab. Da die globale Tiefe mit der lokalen Tiefe des Behälters übereinstimmt, muß das Verzeichnis verdoppelt werden. Anschließend kann `Descartes` in den zuständigen Behälter auf Position `10` eingefügt werden.

Abbildung 4.7 zeigt, daß der Behälter mit `Russel` weiterhin die lokale Tiefe 1 besitzt, also nur das vorderste Bit zur Adressierung verwendet. Bei zwei weiteren Dozenten mit den Personalnummern `2124` bzw. `2128` würde durch das erste Bit ihrer Hash-Werte `00110010001` bzw. `000010110000` der nullte Behälter überlaufen. Da dessen lokale Tiefe 1 kleiner als die globale Tiefe 2 ist, braucht das Verzeichnis nicht verdoppelt zu werden, sondern ein neues Verteilen aller drei mit `0` beginnenden Einträge auf zwei Behälter mit Index `00` und `01` reicht aus.

Werden Daten gelöscht, so können Behälter verschmolzen werden, wenn ihre Inhalte in einem Behälter Platz haben, ihre lokalen Tiefen übereinstimmen und der Wert der ersten $t' - 1$ Bits ihrer Hash-Werte übereinstimmen Dies wäre zum Beispiel in Bild 4.7 der Fall für die Behälter mit Index `10` und `11` nach dem Entfernen von `Kopernikus`. Das Verzeichnis kann halbiert werden, wenn alle lokalen Tiefen kleiner sind als die globale Tiefe t.

4.5 ISAM

Offenes und auch erweiterbares Hashing sind nicht in der Lage, Datensätze in sortierter Reihenfolge auszugeben oder Bereichsabfragen zu bearbeiten. Für Anwendungen, bei denen dies erforderlich ist, kommen Index-Strukturen zum Einsatz (englisch: *index sequential access method = ISAM*) Wir setzen daher voraus, daß sich die Schlüssel der zu verwaltenden Records als Zeichenketten interpretieren lassen und damit eine lexikographische Ordnung auf der Menge der Schlüssel impliziert wird. Sind mehrere Felder am Schlüssel beteiligt, so wird zum Vergleich deren Konkatenation herangezogen.

Neben der Haupt-Datei (englisch: *main file*), die alle Datensätze in lexikographischer Reihenfolge enthält, gibt es nun eine Index-Datei (english: *index file*) mit Verweisen in die Hauptdatei. Die Einträge der Index-Datei sind Tupel, bestehend aus Schlüsseln und Blockadressen, sortiert nach Schlüsseln. Liegt $< v, a >$ in der Index-Datei, so sind alle Record-Schlüssel im Block, auf den a zeigt, größer oder gleich v. Zur Anschauung: Fassen wir ein Telefonbuch als Hauptdatei auf (eine Seite $\equiv$ ein Block), so bilden alle die Namen, die jeweils links oben auf den Seiten vermerkt sind, einen Index. Da im Index nur ein Teil der Schlüssel aus der Hauptdatei zu finden sind, spricht man von einer dünn besetzten Index-Datei (englisch: *sparse index*).

Wir nehmen an, die Records seien verschiebbar und pro Block sei im Header vermerkt, welche Sub-blocks belegt sind. Dann ergeben sich die folgenden Operationen:

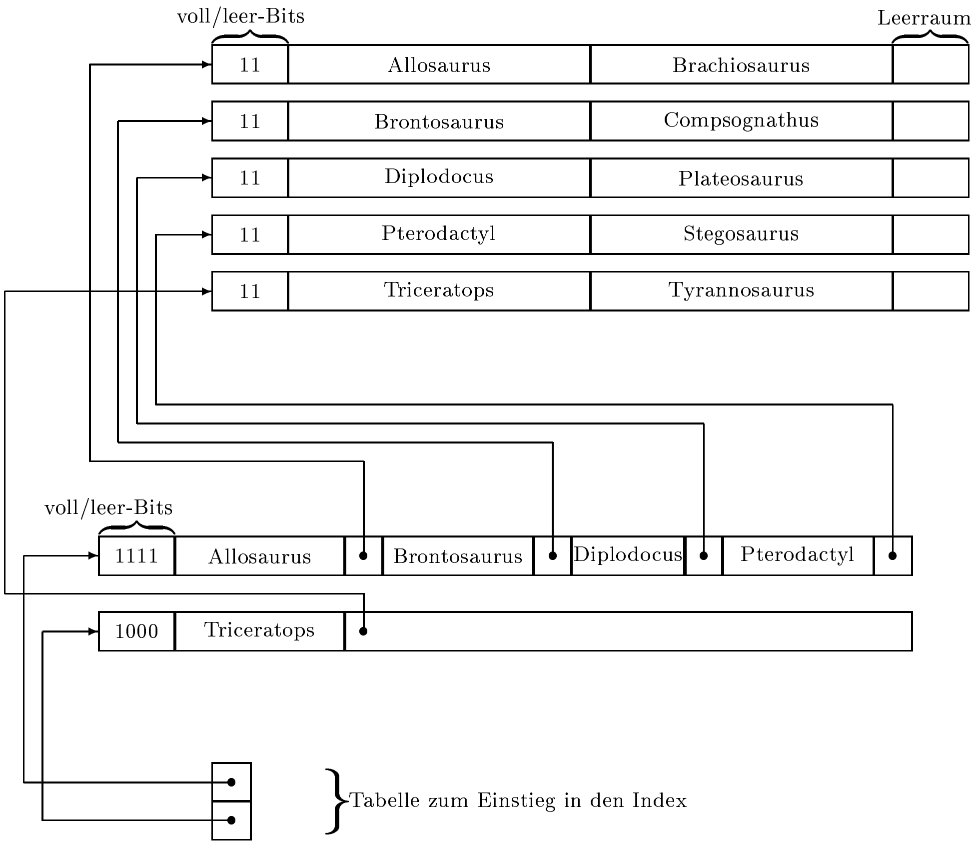

Abbildung 4.8: ISAM: Ausgangslage

- **LOOKUP:**
  Gesucht wird ein Record mit Schlüssel v_1. Suche (mit binary search) in der Index-Datei den letzten Block mit erstem Eintrag $v_2 \leq v_1$. Suche in diesem Block das letzte Paar (v_3, a) mit $v_3 \leq v_1$. Lies Block mit Adresse a und durchsuche ihn nach Schlüssel v_1.

- **MODIFY:**
  Führe zunächst LOOKUP durch. Ist der Schlüssel an der Änderung beteiligt, so wird MODIFY wie ein DELETE mit anschließendem INSERT behandelt. Andernfalls kann das Record überschrieben und dann der Block zurückgeschrieben werden.

- **INSERT:**
  Eingefügt wird ein Record mit Schlüssel v. Suche zunächst mit LOOKUP den Block B_i, auf dem v zu finden sein müßte (falls v kleinster Schlüssel, setze i =1). Falls B_i nicht vollständig gefüllt ist: Füge Record in B_i an passender Stelle ein, und verschiebe ggf. Records um eine Position nach rechts (Full/Empty-Bits korrigieren). Wenn v kleiner als alle bisherigen Schlüssel ist, so korrigiere Index-Datei. Wenn B_i gefüllt ist: Überprüfe, ob B_{i+1} Platz hat. Wenn ja: Schiebe überlaufendes Record nach B_{i+1} und korrigiere Index. Wenn nein: Fordere neuen Block B_i' an, speichere das Record dort, und füge im Index einen Verweis ein.

- **DELETE:** analog zu INSERT

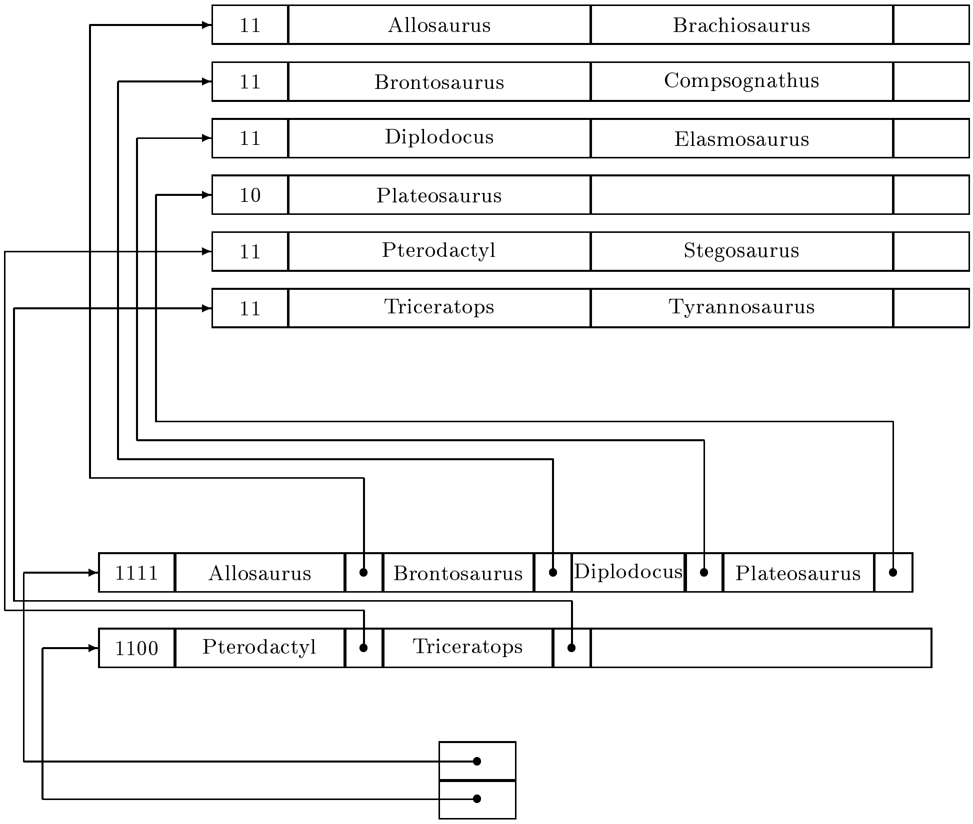

Abbildung 4.9: ISAM: nach Einfügen von Elasmosaurus

Bemerkung: Ist die Verteilung der Schlüssel bekannt, so sinkt für n Index-Blöcke die Suchzeit durch *Interpolation Search* auf $log\ log\ n$ Schritte!

Abbildung 4.8 zeigt die Ausgangslage für eine Hauptdatei mit Blöcken, die jeweils 2 Records speichern können. Die Blöcke der Index-Datei enthalten jeweils vier Schlüssel/Adreß-Paare. Weiterhin gibt es im Hauptspeicher eine Tabelle mit Verweisen zu den Index-Datei-Blöcken.

Abbildung 4.9 zeigt die Situation nach dem Einfügen von `Elasmosaurus`. Hierzu findet man zunächst als Einstieg `Diplodocus`. Der zugehörige Dateiblock ist voll, so daß nach Einfügen von `Elasmosaurus` für das überschüssige Record `Plateosaurus` ein neuer Block angelegt und sein erster Schlüssel in die Index-Datei eingetragen wird.

Nun wird `Brontosaurus` umbenannt in `Apatosaurus`. Hierzu wird zunächst `Brontosaurus` gelöscht, sein Dateinachfolger `Compsognathus` um einen Platz vorgezogen und der Schlüssel in der Index-Datei, der zu diesem Blockzeiger gehört, modifiziert. Das Einfügen von `Apatosaurus` bewirkt einen Überlauf von `Brachiosaurus` in den Nachfolgeblock, in dem `Compsognathus` nun wieder an seinen alten Platz rutscht. Im zugehörigen Index-Block verschwindet daher sein Schlüssel wieder und wird überschrieben mit `Brachiosaurus`.

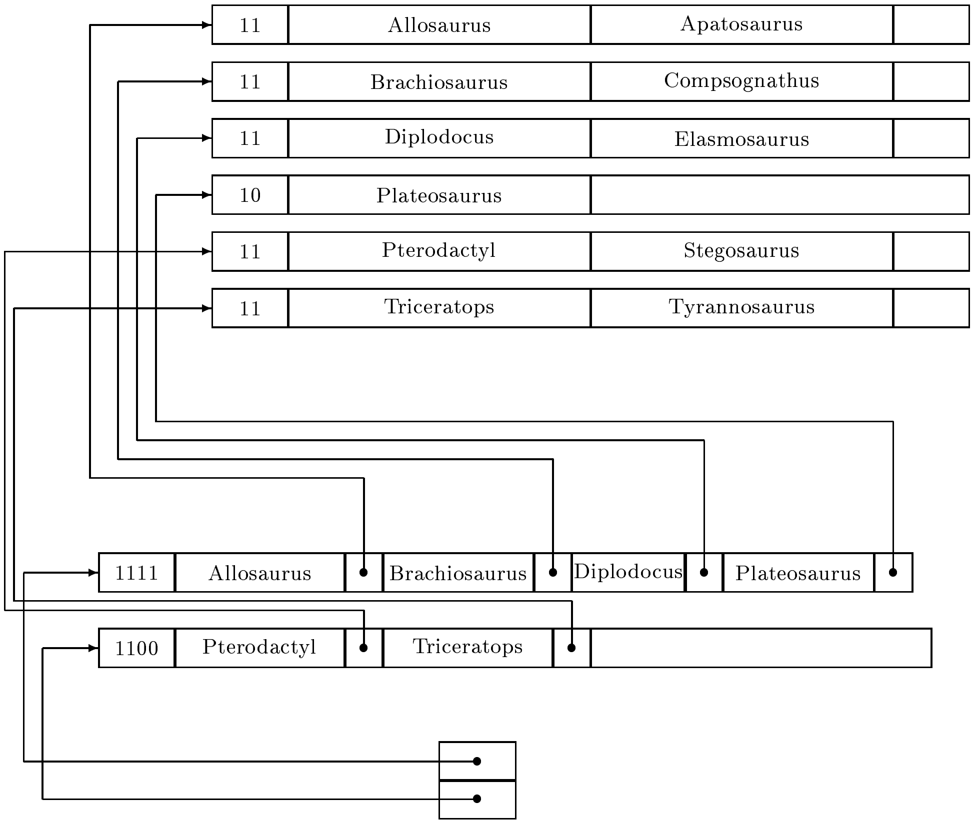

Abbildung 4.10: ISAM: nach Umbenennen von Brontosaurus

4.6 B*-Baum

Betrachten wir das Index-File als Daten-File, so können wir dazu ebenfalls einen weiteren Index konstruieren und für dieses File wiederum einen Index usw. Diese Idee führt zum B*-Baum.

Ein B*-Baum mit Parameter k wird charakterisiert durch folgende Eigenschaften:

- Jeder Weg von der Wurzel zu einem Blatt hat dieselbe Länge.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k Nachfolger.
- Jeder Knoten hat höchstens $2 \cdot k$ Nachfolger.
- Die Wurzel hat keinen oder mindestens 2 Nachfolger.

Der Baum T befindet sich im Hintergrundspeicher, und zwar nimmt jeder Knoten einen Block ein. Ein Knoten mit j Nachfolgern speichert j Paare von Schlüsseln und Adressen $(s_1,\ a_1), \ldots, (s_j,\ a_j)$. Es gilt $s_1\ \leq s_2\ \leq\ \ldots\ \leq s_j$. Eine Adresse in einem Blattknoten bezeichnet den Datenblock mit den restlichen Informationen zum zugehörigen Schlüssel, sonst bezeichnet sie den Block zu einem Baumknoten: Enthalte der Block für Knoten p die Einträge $(s_1, a_1), \ldots, (s_j, a_j)$. Dann ist der erste

Schlüssel im i-ten Sohn von p gleich s_i, alle weiteren Schlüssel in diesem Sohn (sofern vorhanden) sind größer als s_i und kleiner als s_{i+1}.

Wir betrachten nur die Operationen auf den Knoten des Baumes und nicht auf den eigentlichen Datenblöcken. Gegeben sei der Schlüssel s.

LOOKUP: Beginnend bei der Wurzel steigt man den Baum hinab in Richtung des Blattes, welches den Schlüssel s enthalten müßte. Hierzu wird bei einem Knoten mit Schlüsseln $s_1, s_2, \ldots, s_j$ als nächstes der i-te Sohn besucht, wenn gilt $s_i \leq s < s_{i+1}$.

MODIFY: Wenn das Schlüsselfeld verändert wird, muß ein DELETE mit nachfolgendem INSERT erfolgen. Wenn das Schlüsselfeld nicht verändert wird, kann der Datensatz nach einem LOOKUP überschrieben werden.

INSERT: Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthalten soll. Wenn B weniger als $2k$ Einträge hat, so wird s eingefügt, und es werden die Vorgängerknoten berichtigt, sofern s kleinster Schlüssel im Baum ist. Wenn B $2 \cdot k$ Einträge hat, wird ein neues Blatt B' generiert, mit den größeren k Einträgen von B gefüllt und dann der Schlüssel s eingetragen. Der Vorgänger von B und B' wird um einen weiteren Schlüssel s' (kleinster Eintrag in B') erweitert. Falls dabei Überlauf eintritt, pflanzt sich dieser nach oben fort.

DELETE: Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthält. Das Paar (s, a) wird entfernt und ggf. der Schlüsseleintrag der Vorgänger korrigiert. Falls B jetzt $k-1$ Einträge hat, wird der unmittelbare Bruder B' mit den meisten Einträgen bestimmt. Haben beide Brüder gleich viel Einträge, so wird der linke genommen. Hat B' mehr als k Einträge, so werden die Einträge von B und B' auf diese beiden Knoten gleichmäßig verteilt. Haben B und B' zusammen eine ungerade Anzahl, so erhält der linke einen Eintrag mehr. Hat B' genau k Einträge, so werden B und B' verschmolzen. Die Vorgängerknoten müssen korrigiert werden.

Abbildung 4.11 zeigt das dynamische Verhalten eines B*-Baums mit dem Parameter $k = 2$. Es werden nacheinander die Schlüssel `3, 7, 1, 16, 4, 14, 12, 6, 2, 15, 13, 8, 10, 5, 11, 9` eingefügt und insgesamt 8 Schnappschüsse zu folgenden Zeitpunkten gezeichnet:

```
3,7,1,16,4,14,12,6,2,15,13,8,10,5,11,9
          ↑ ↑       ↑ ↑  ↑   ↑  ↑      ↑
          1.2.      3.4. 5.  6. 7.     8.
```

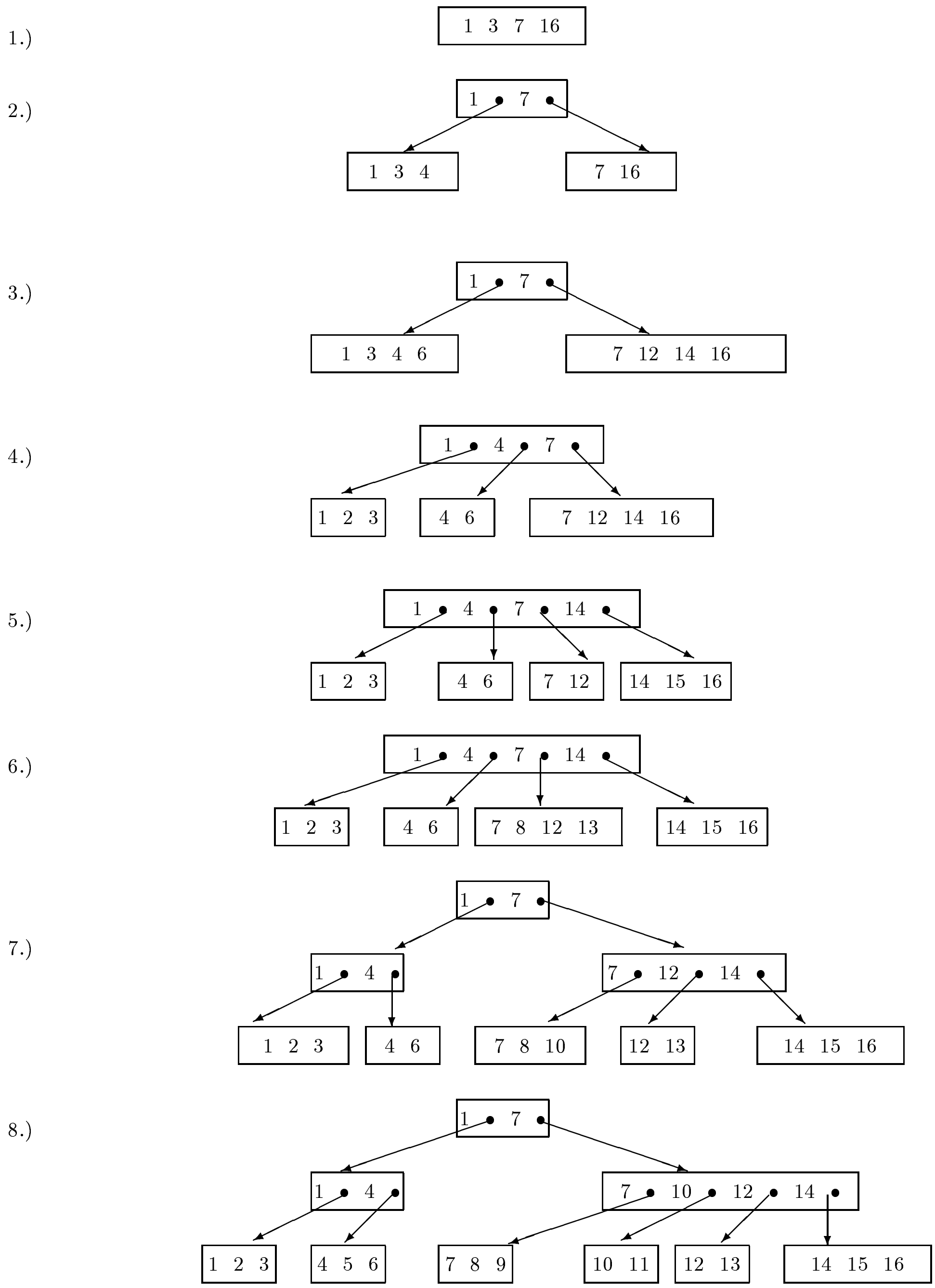

Abbildung 4.11: dynamisches Verhalten eines B*-Baums

Der Parameter k ergibt sich aus dem Platzbedarf für die Schlüssel/Adreßpaare und der Blockgröße. Die Höhe des Baumes ergibt sich aus der benötigten Anzahl von Verzweigungen, um in den Blättern genügend Zeiger auf die Datenblöcke zu haben.

Beispiel für die Berechnung des Platzbedarfs eines B*-Baums:

Gegeben seien 300.000 Datenrecords à 100 Bytes. Jeder Block umfasse 1.024 Bytes. Ein Schlüssel sei 15 Bytes lang, eine Adresse bestehe aus 4 Bytes.

Daraus errechnet sich der Parameter k wie folgt

$$\lfloor \frac{1024}{15+4} \rfloor = 53 \Rightarrow k = 26$$

Die Wurzel sei im Mittel zu 50 % gefüllt (hat also 26 Söhne), ein innerer Knoten sei im Mittel zu 75 % gefüllt (hat also 39 Söhne), ein Datenblock sei im Mittel zu 75 % gefüllt (enthält also 7 bis 8 Datenrecords). 300.000 Records sind also auf $\lfloor \frac{300.000}{7,5} \rfloor = 40.000$ Datenblöcke verteilt.

Die Zahl der Zeiger entwickelt sich daher auf den oberen Ebenen des Baums wie folgt:

Höhe	Anzahl Knoten	Anzahl Zeiger		
0	1	26		
1	26	$26 \cdot 39$	=	1.014
2	$26 \cdot 39$	$26 \cdot 39 \cdot 39$	=	39.546

Damit reicht die Höhe 2 aus, um genügend Zeiger auf die Datenblöcke bereitzustellen. Der Platzbedarf beträgt daher

$$1 + 26 + 26 \cdot 39 + 39546 \approx 40.000 Blöcke.$$

Das LOOKUP auf ein Datenrecord verursacht also vier Blockzugriffe: es werden drei Indexblöcke auf Ebene 0, 1 und 2 sowie ein Datenblock referiert. Zum Vergleich: Das Heapfile benötigt 30.000 Blöcke.

Soll für offenes Hashing eine mittlere Zugriffszeit von 4 Blockzugriffen gelten, so müssen in jedem Bucket etwa 5 Blöcke sein (1 Zugriff für Hash-Directory, 3 Zugriffe im Mittel für eine Liste von 5 Blöcken). Von diesen 5 Blöcken sind 4 voll, der letzte halbvoll. Da 10 Records in einen Datenblock passen, befinden sich in einem Bucket etwa $4,5 \cdot 10 = 45$ Records. Also sind $\frac{300.000}{45} = 6.666$ Buckets erforderlich. Da 256 Adressen in einen Block passen, werden $\lfloor \frac{6666}{256} \rfloor = 26$ Directory-Blöcke benötigt. Der Platzbedarf beträgt daher $26 + 5 \cdot 6666 = 33356$.

Zur Bewertung von B*-Bäumen läßt sich sagen:

- **Vorteile:** dynamisch, schnell, Sortierung generierbar (ggf. Blätter verzeigern).
- **Nachteile:** komplizierte Operationen, Speicheroverhead.

4.7 Sekundär-Index

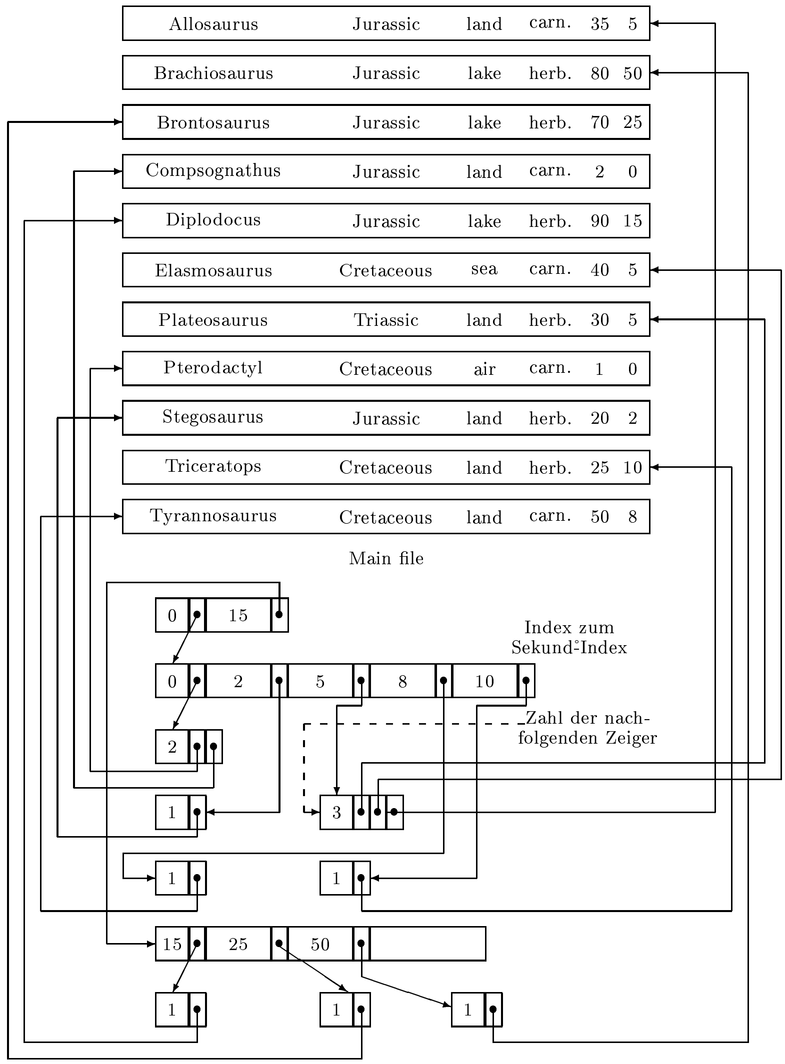

Abbildung 4.12: Sekundär-Index für GEWICHT

Die bisher behandelten Organisationsformen sind geeignet zur Suche nach einem Record, dessen Schlüssel gegeben ist. Um auch effizient Nicht-Schlüssel-Felder zu behandeln, wird für jedes Attribut, das unterstützt werden soll, ein sogenannter Sekundär-Index (englisch: *secondary index*) angelegt. Er besteht aus einem Index-File mit Einträgen der Form $<$Attributwert, Adresse$>$.

Abbildung 4.12 zeigt für das Dinosaurier-File einen *secondary index* für das Attribut GEWICHT, welches, gespeichert in der letzten Record-Komponente, von 5 bis 50 variiert. Der Sekundär-Index (er wird erreicht über einen Index mit den Einträgen 0 und 15) besteht aus den Blöcken $<$0, 2, 5, 8, 10$>$ und $<$15, 25, 50$>$. Die beim Gewicht g gespeicherte Adresse führt zunächst zu einem Vermerk zur Anzahl der Einträge mit dem Gewicht g und dann zu den Adressen der Records mit Gewicht g.

Kapitel 5

Mehrdimensionale Suchstrukturen

5.1 Problemstellung

Ein Sekundär-Index ist in der Lage, alle Records mit $x_1 \leq a \leq x_2$ zu finden. Nun heißt die Aufgabe: Finde alle Records mit $x_1 \leq a_1 \leq x_2$ und $y_1 \leq a_2 \leq y_2, \ldots$

Beispiel für mehrdimensionale Bereichsabfrage:
Gesucht sind alle Personen mit der Eigenschaft

Alter	zwischen 20 und 30 Jahre alt
Einkommen	zwischen 2000 und 3000 DM
PLZ	zwischen 40000 und 50000

Im folgenden betrachten wir (wegen der einfacheren Veranschaulichung) nur den 2-dimensionalen Fall. Diese Technik ist auf beliebige Dimensionen verallgemeinerbar.

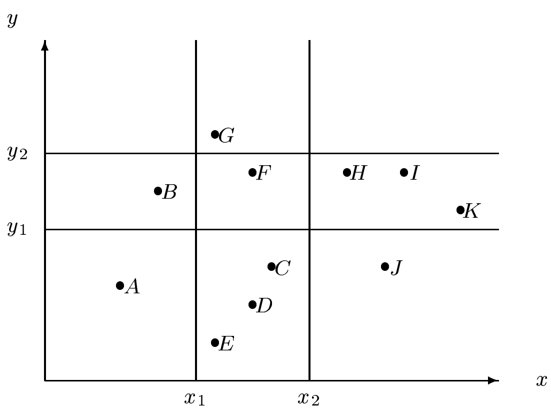

Abbildung 5.1: Fläche mit Datenpunkten

Abbildung 5-1 zeigt eine zweidimensionale Fläche mit Datenpunkten sowie ein Query-Rechteck, gegeben durch vier Geraden.

Die Aufgabe besteht darin, alle Punkte zu ermitteln, die im Rechteck liegen. Hierzu bieten sich zwei naheliegende Möglichkeiten an:

- Projektion durchführen auf x oder y mit binärer Suche über vorhandenen Index, danach sequentiell durchsuchen, d.h. zunächst werden G, F, C, D, E ermittelt, danach bleibt F übrig
- Projektion durchführen auf x und Projektion durchführen auf y, anschließend Durchschnitt bilden.

Es ist offensichtlich, daß trotz kleiner Trefferzahl ggf. lange Laufzeiten auftreten können. Dagegen ist für die 1-dimensionale Suche bekannt: Der Aufwand beträgt O($k + \log n$) bei k Treffern in einem Suchbaum mit n Knoten.

5.2 k-d-Baum

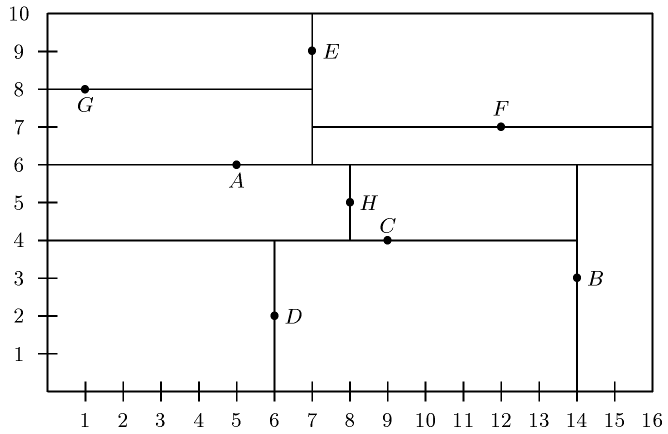

Abbildung 5.2: Durch die Datenpunkte A,B,C,D,E,F,G,H partitionierte Fläche

Eine Verallgemeinerung eines binären Suchbaums mit einem Sortierschlüssel bildet der *k-d-Baum* mit k-dimensionalem Sortierschlüssel. Er verwaltet eine Menge von mehrdimensionalen Datenpunkten, wie z.B. Abbildung 5.2 für den 2-dimensionalen Fall zeigt. In der homogenen Variante enthält jeder Baumknoten ein komplettes Datenrecord und zwei Zeiger auf den linken und rechten Sohn (Abbildung 5.3). In der inhomogenen Variante enhält jeder Baumknoten nur einen Schlüssel und die Blätter verweisen auf die Datenrecords (Abbildung 5.4). In beiden Fällen werden die Werte der einzelnen Attribute abwechselnd auf jeder Ebene des Baumes zur Diskriminierung verwendet. Es handelt sich um eine statische Struktur; die Operationen Löschen und die Durchführung einer Balancierung sind sehr aufwendig.

Im 2-dimensionalen Fall gilt für jeden Knoten mit Schlüssel $[x/y]$:

	im linken Sohn	im rechten Sohn
auf ungerader Ebene	alle Schlüssel $\leq x$	alle Schlüssel $> x$
auf gerader Ebene	alle Schlüssel $\leq y$	alle Schlüssel $> y$

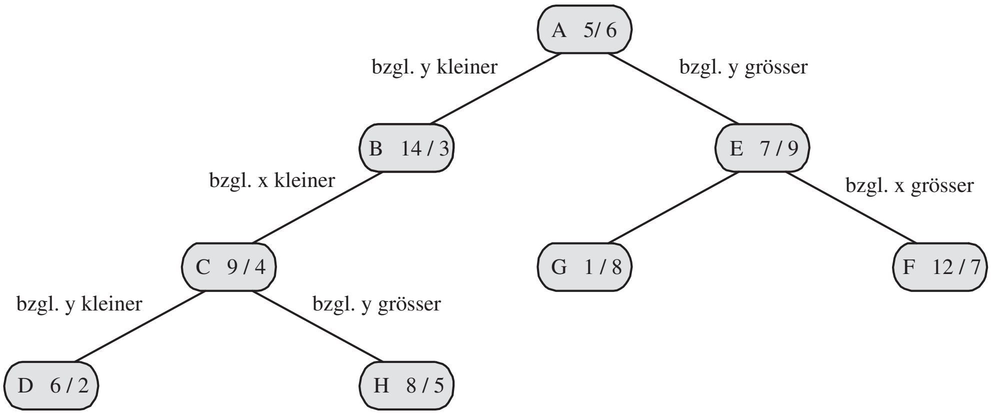

Abbildung 5.3: 2-d-Baum (homogen) zu den Datenpunkten A,B,C,D,E,F,G,H

Die Operationen auf einem $2-d$-Baum laufen analog zum binärem Baum ab:

- **Insert**:
  Suche mit Schlüssel $[x/y]$ unter Abwechslung der Dimension die Stelle, wo der $[x/y]$-Knoten sein müßte und hänge ihn dort ein.

- **Exakt Match** (z.B. finde Record $[15/5]$):
  Suche mit Schlüssel $[x/y]$ unter Abwechslung der Dimension bis zu der Stelle, wo der $[x/y]$-Knoten sein müßte.

- **Partial Match** (z.B. finde alle Records mit $x = 7$):
  An den Knoten, an denen nicht bzgl. x diskriminiert wird, steige in beide Söhne ab; an den Knoten, an denen bzgl. x diskriminiert wird, steige in den zutreffenden Teilbaum ab.

- **Range-Query** (z.B. finde alle Records $[x, y]$ mit $7 \leq x \leq 13, 5 \leq y \leq 8$):
  An den Knoten, an denen die Diskriminatorlinie das Suchrechteck schneidet, steige in beide Söhne ab, sonst steige in den zutreffenden Sohn ab. Beobachtung: Laufzeit $k + \log n$ Schritte bei k Treffern!

- **Best-Match** (z.B. finde nächstgelegenes Record zu $x = 7, y = 3$):
  Dies entspricht einer Range-Query, wobei statt eines Suchrechtecks jetzt ein Suchkreis mit Radius gemäß Distanzfunktion vorliegt. Während der Baumtraversierung schrumpft der Suchradius. Diese Strategie ist erweiterbar auf k-best-Matches.

Bei der inhomogenen Variante enthalten die inneren Knoten je nach Ebene die Schlüsselinformation der zuständigen Dimension sowie Sohnzeiger auf weitere innere Knoten. Nur die Blätter verweisen auf Datenblöcke der Hauptdatei, die jeweils mehrere Datenrecords aufnehmen können. Auch die inneren Knoten werden zu Blöcken zusammengefaßt, wie auf Abbildung 5.5 zu sehen ist. In Abbildung 5.4 befinden sich z.B. die Datenrecords *C, B* und *D* in einem Block.

Abbildung 5.6 zeigt, daß neben der oben beschriebenen 2-d-Baum-Strategie eine weitere Möglichkeit existiert, den Datenraum zu partitionieren. Dies führt zu den sogenannten *Gitterverfahren*.

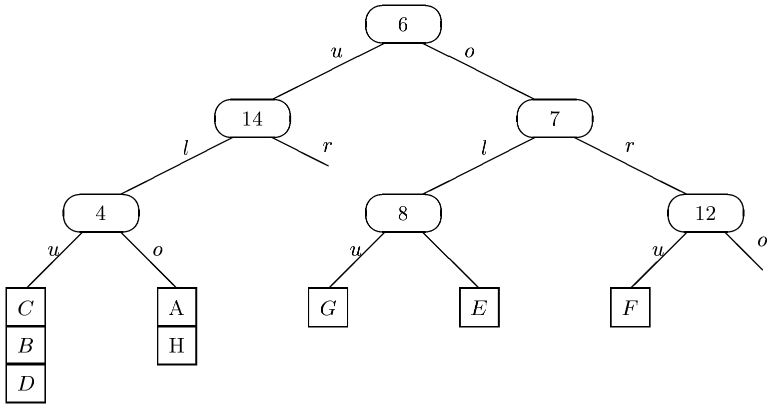

Abbildung 5.4: 2-d-Baum (inhomogen) zu den Datenpunkten A,B,C,D,E,F,G,H

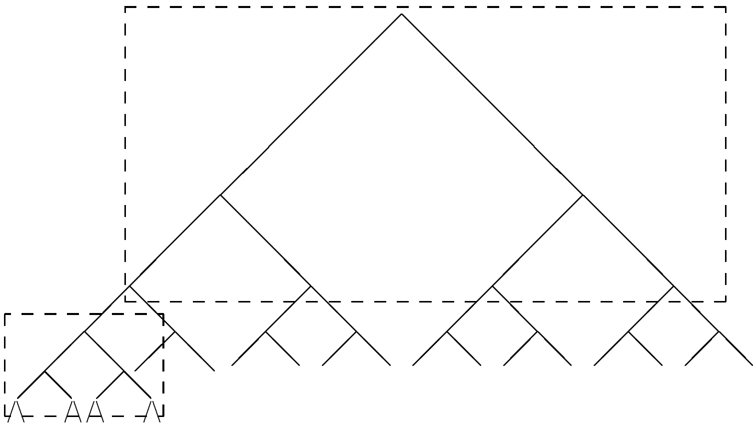

Abbildung 5.5: Zusammenfassung von je 7 inneren Knoten auf einem Index-Block

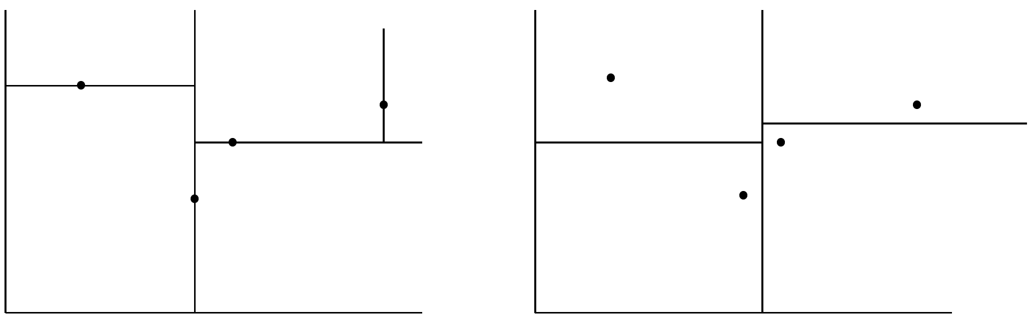

Abbildung 5.6: Partitionierungsmöglichkeiten des Raumes

5.3 Gitterverfahren mit konstanter Gittergröße

Gitterverfahren, die mit konstanter Gittergröße arbeiten, teilen den Datenraum in Quadrate fester Größe auf. Abbildung 5.7 zeigt eine Anordnung von 24 Datenblöcken, die jeweils eine feste Anzahl von Datenrecords aufnehmen können. Über einen Index werden die Blöcke erreicht. Diese statische Partitionierung lastet die Datenblöcke natürlich nur bei einer Gleichverteilung wirtschaftlich aus und erlaubt bei Ballungsgebieten keinen effizienten Zugriff.

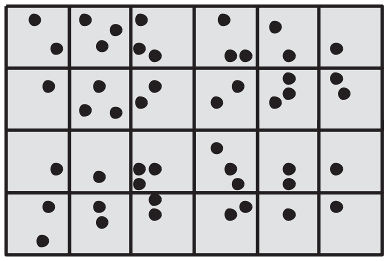

Abbildung 5.7: Partitionierung mit fester Gittergröße

5.4 Grid File

Als Alternative zu den Verfahren mit fester Gittergröße stellten Hinrichs und Nievergelt im Jahre 1981 das *Grid File* vor, welches auch bei dynamisch sich änderndem Datenbestand eine *2-Platten-Zugriffsgarantie* gibt.

Erreicht wird dies (bei k-dimensionalen Tupeln) durch

- k Skalen zum Einstieg ins Grid-Directory (im Hauptspeicher)
- Grid-Direktory zum Finden der Bucket-Nr. (im Hintergrundspeicher)
- Buckets für Datensätze (im Hintergrundspeicher)

Zur einfacheren Veranschaulichung beschreiben wir die Technik für Dimension $k = 2$. Verwendet werden dabei

- **zwei eindimensionale Skalen**,
  welche die momentane Unterteilung der X- bzw. Y-Achse enthalten:

```
var X: array [0..max_x] of attribut_wert_x;
var Y: array [0..max_y] of attribut_wert_y;
```

- **ein 2-dimensionales Grid-Directory**,
  welches Verweise auf die Datenblöcke enthält:

```
var G: array [0..max_x - 1, 0..max_y - 1] of pointer;
```

D.h. `G[i, j]` enthält eine Bucketadresse, in der ein rechteckiger Teilbereich der Datenpunkte abgespeichert ist. Zum Beispiel sind alle Punkte mit $30 < x \leq 40, 2050 < y \leq 2500$ im Bucket mit Adresse `G[1,2]` zu finden (in Abbildung 5.8 gestrichelt umrandet). Achtung: mehrere Gitterzellen können im selben Bucket liegen.

- **mehrere Buckets**,
  welche jeweils eine maximale Zahl von Datenrecords aufnehmen können.

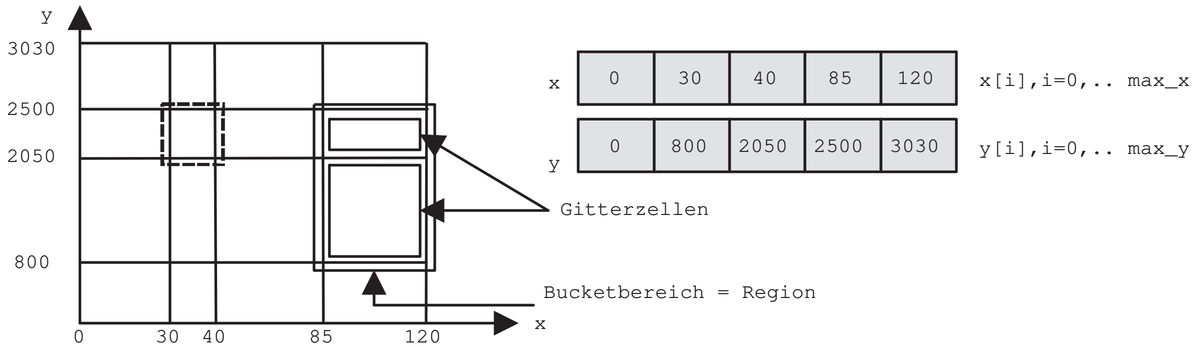

Abbildung 5.8: Skalen und resultierende Gitterzellen

Beispiel für ein Lookup mit $x = 100, y = 1000$:
Suche in Skala x den letzten Eintrag $< x$. Er habe den Index $i = 3$.
Suche in Skala y den letzten Eintrag $< y$. Er habe den Index $j = 1$.
Lade den Teil des Grid-Directory in den Hauptspeicher, der $G[3, 1]$ enthält.
Lade Bucket mit Adresse $G[3, 1]$.

Beispiel für den Zugriff auf das Bucket-Directory:
Vorhanden seien 1.000.000 Datentupel, jeweils 4 passen in einen Block. Die X- und die Y-Achse habe jeweils 500 Unterteilungen. Daraus ergeben sich 250.000 Einträge für das Bucket-Directory G. Bei 4 Bytes pro Zeiger und 1024 Bytes pro Block passen 250 Zeiger auf einen Directory-Block. Also gibt es 1000 Directory-Blöcke. D.h. $G[i, j]$ findet sich auf Block $2 \cdot j$ als i-te Adresse, falls $i < 250$ und befindet sich auf Block $2 \cdot j + 1$ als $(i - 250)$-te Adresse, falls $i \geq 250$

Bei einer *range query*, gegeben durch ein Suchrechteck, werden zunächst alle Gitterzellen bestimmt, die in Frage kommen, und dann die zugehörenden Buckets eingelesen.

5.5 Aufspalten und Mischen beim Grid File

Die grundsätzliche Idee besteht darin, bei sich änderndem Datenbestand durch Modifikation der Skalen die Größen der Gitterzellen anzupassen.

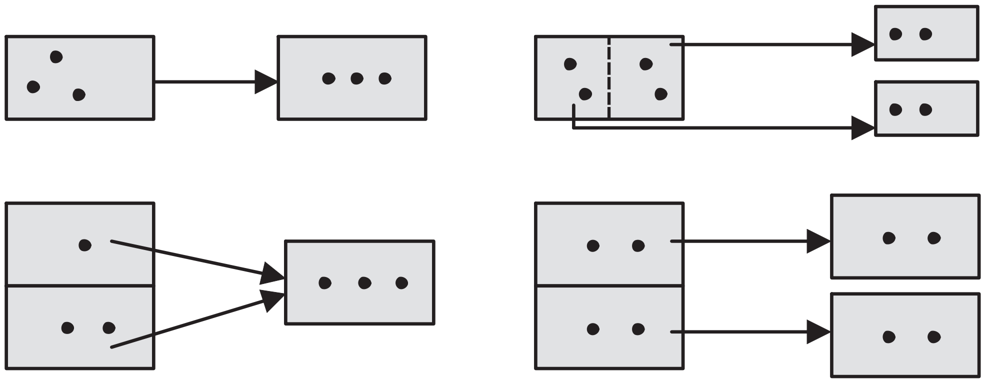

Abbildung 5.9: Konsequenzen eines Bucket-Überlauf (mit und ohne Gitterverfeinerung)

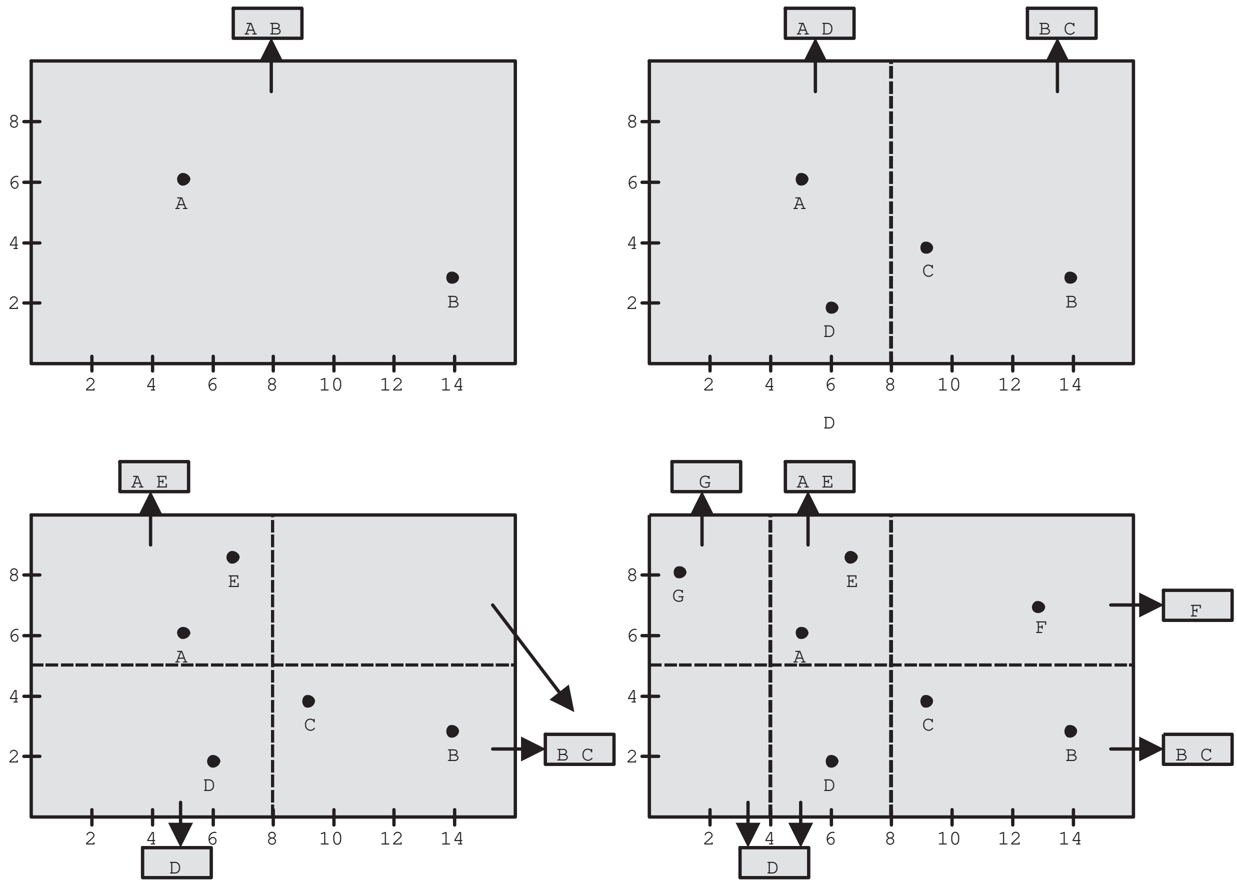

Abbildung 5.10: Aufspalten der Regionen für Datenpunkte A, B, C, D, E, F, G

Aufspalten von Regionen

Der Überlauf eines Buckets, dessen Region aus einer Zelle besteht, verursacht eine Gitterverfeinerung, die gemäß einer *Splitting Policy* organisiert wird. Im wesentlichen wird unter Abwechslung der Dimension die Region halbiert. Dieser Sachverhalt wird in der oberen Hälfte von Abbildung 5.9

demonstriert unter der Annahme, daß drei Datenrecords in ein Datenbucket passen. In der unteren Hälfte von Abbildung 5.9 ist zu sehen, daß bei Überlauf eines Buckets, dessen Region aus mehreren Gitterzellen besteht, keine Gitterverfeinerung erforderlich ist.

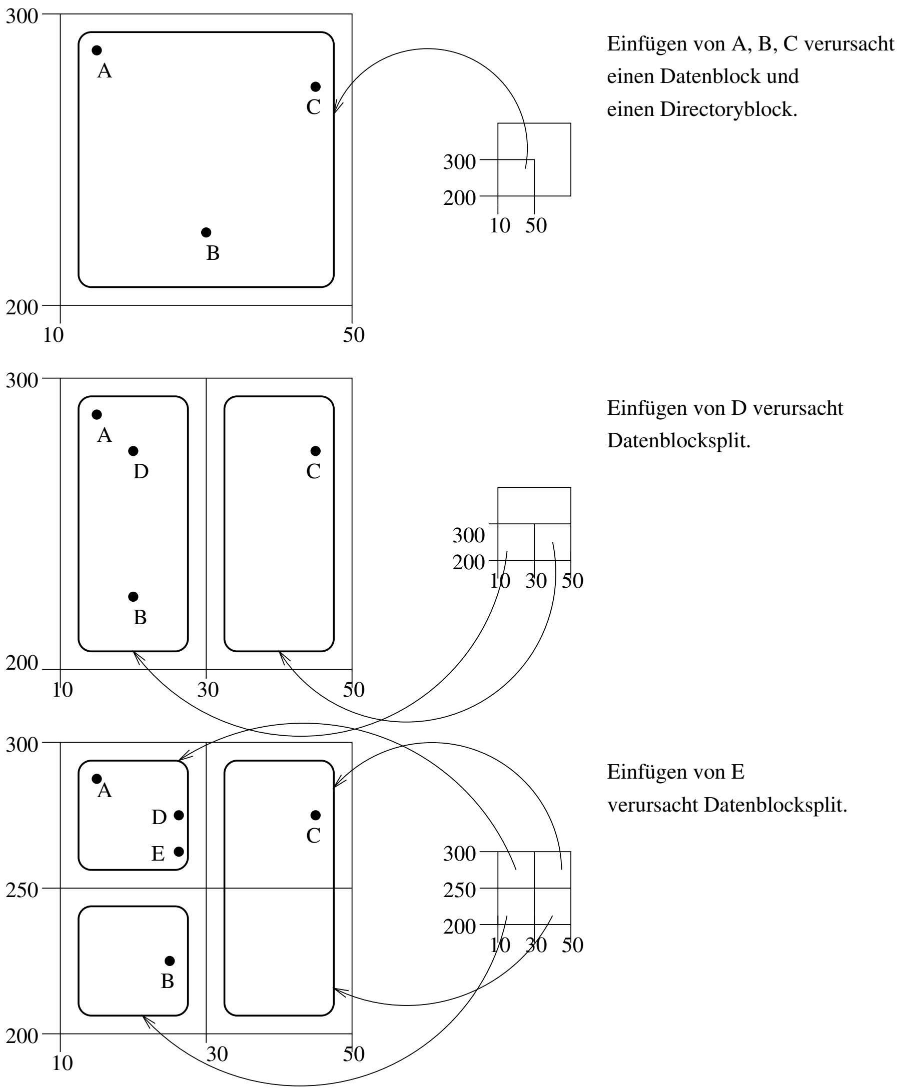

Abbildung 5.11: Dynamik des Grid File beim Einfügen der Datenpunkte A,B,C,D,E

Abbildung 5.10 zeigt die durch das sukzessive Einfügen in ein Grid File entwickelte Dynamik. Es handelt sich dabei um die in Abbildung 4.14 verwendeten Datenpunkte A, B, C, D, E, F, G. In dem Beispiel wird angenommen, daß 2 Datenrecords in einen Datenblock passen. Bei überlaufendem Datenblock wird die Region halbiert, wobei die Dimension abwechselt. Schließlich hat das Grid-Directory 6 Zeiger auf insgesamt 5 Datenblöcke. Die x-Skala hat drei Einträge, die y-Skala hat zwei Einträge.

Zu der dynamischen Anpassung der Skalen und Datenblöcke kommt noch die Buchhaltung der Directory-Blöcke. Dies wird in der Abbildung 5.11 demonstriert anhand der (neu positionierten) Datenpunkte A, B, C, D, E. Von den Directory-Blöcken wird angenommen, daß sie vier Adressen speichern können, in einen Datenblock mögen drei Datenrecords passen. Grundsätzlich erfolgt der Einstieg in den zuständi-

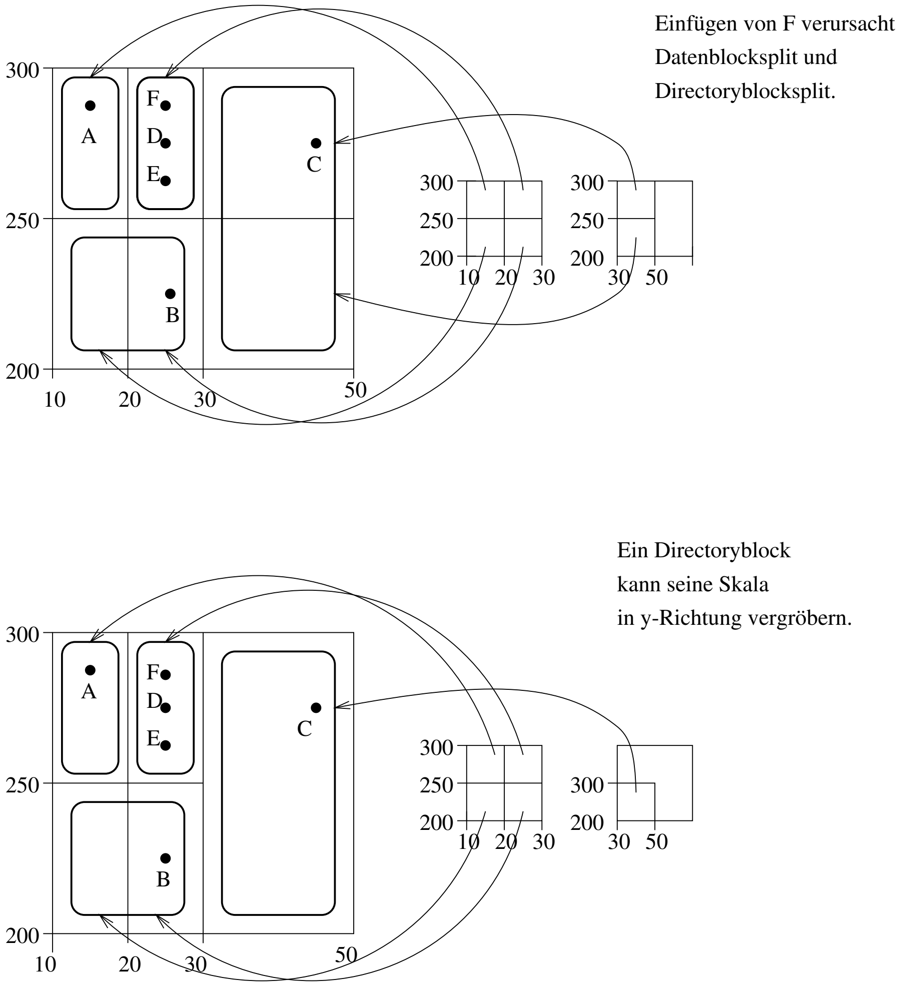

Abbildung 5.12: Vergröberung des Grid Directory nach Aufspalten

gen Directory-Block über das sogenannte *Root-Directory*, welches im Hauptspeicher mit vergröberten Skalen liegt. Die durch das Einfügen verursachte Aufspaltung eines Datenblocks und die dadurch ausgelösten Verfeinerungen der Skalen ziehen auch Erweiterungen im Directory-Block nach. Abbildung 5.12 zeigt, wie beim Überlauf eines Directory-Blockes dieser halbiert und auf zwei Blöcke verteilt wird. Dabei kommt es zu einer Vergröberung der Skala.

Mischen von Regionen

Die beim Expandieren erzeugte Neustrukturierung bedarf einer Umordnung, wenn der Datenbestand schrumpft, denn nach dem Entfernen von Datenrecords können Datenblöcke mit zu geringer Auslastung entstehen, welche dann zusammengefaßt werden sollten. Die *Merging Policy* legt den Mischpartner und den Zeitpunkt des Mischens fest:

- Mischpartner zu einem Bucket X kann nur ein Bucket Y sein, wenn die Vereinigung der beiden Bucketregionen ein Rechteck bilden (Abbildung 5.13). Grund: Zur effizienten Bearbeitung von Range-Queries sind nur rechteckige Gitter sinnvoll!

- Das Mischen wird ausgelöst, wenn ein Bucket höchstens zu 30 % belegt ist und wenn das vereinigte Bucket höchstens zu 70 % belegt sein würde (um erneutes Splitten zu vermeiden)

Abbildung 5.13: Zusammenfassung von Regionen

5.6 Verwaltung geometrischer Objekte

In der bisherigen Anwendung repräsentierten die Datenpunkte im k-dimensionale Raum k-stellige Attributkombinationen. Wir wollen jetzt mithilfe der Datenpunkte geometrische Objekte darstellen und einfache geometrische Anfragen realisieren.

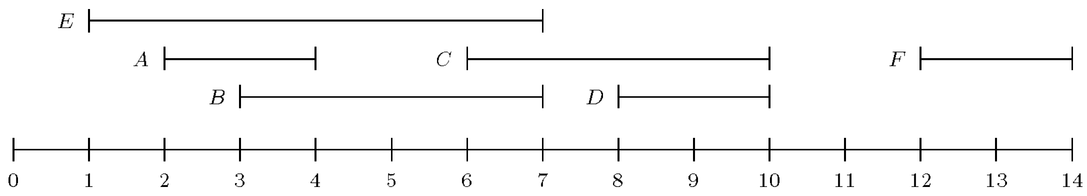

Abbildung 5.14: Intervalle A,B,C,D,E,F über der Zahlengeraden

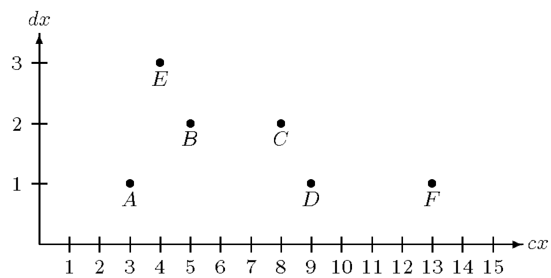

Abbildung 5.15: Repräsentation von Intervallen durch Punkte

Abbildung 5.14 zeigt eine Ansammlung von Intervallen, die zu verwalten seien. Die Intervalle sollen durch Punkte im mehrdimensionalen Raum dargestellt werden. Wenn alle Intervalle durch ihre Anfangs- und Endpunkte repräsentiert würden, kämen sie auf der Datenfläche nur oberhalb der 45-Grad-Geraden zu liegen.

Abbildung 5.15 präsentiert eine wirtschaftlichere Verteilung, indem jede Gerade durch ihren Mittelpunkt und ihre halbe Länge repräsentiert wird.

Typische Queries an die Intervall-Sammlung lauten:

- Gegeben Punkt P, finde alle Intervalle, die ihn enthalten.
- Gegeben Intervall I, finde alle Intervalle, die es schneidet.

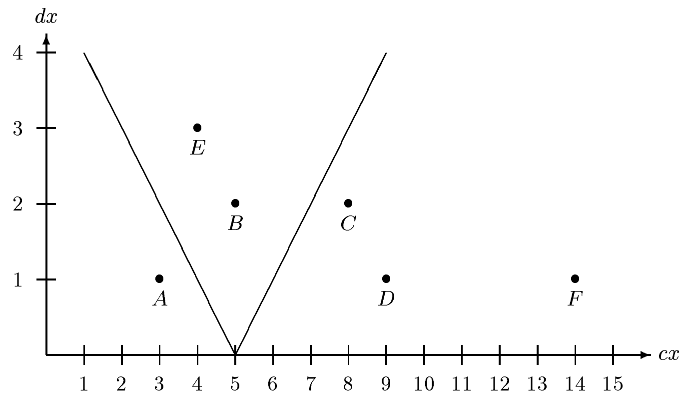

Abbildung 5.16: Abfragekegel zu Punkt p=5

Abbildung 5.16 zeigt den kegelförmigen Abfragebereich zum Query-Punkt p=5, in dem alle Intervalle (repräsentiert durch Punkte) liegen, die den Punkt p enthalten. Grundlage ist die Überlegung, daß ein Punkt P genau dann im Intervall mit Mitte m und halber Länge d enhalten ist, wenn gilt: $m - d \leq p \leq m + d$

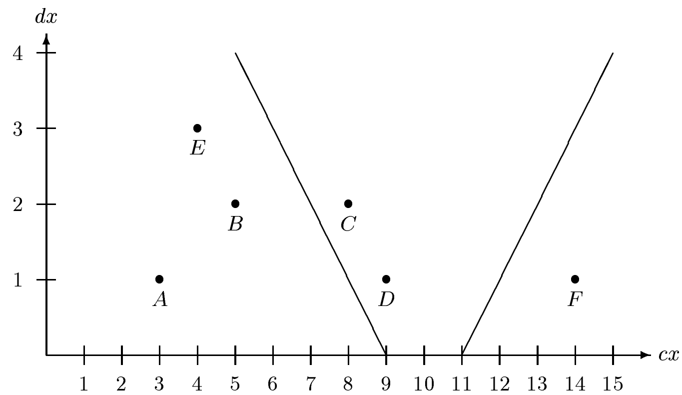

Abbildung 5.17: Abfragekegel zu Intervall mit Mitte s=10 und halber Länge t=1

Abbildung 5.17 zeigt den kegelförmigen Abfragebereich zu dem Query-Intervall mit Mittelpunkt s=10 und halber Länge t=1, in dem alle Intervalle (repäsentiert durch Punkte) liegen, die das Query-Intervall schneiden. Grundlage ist die Überlegung, daß ein Intervall mit Mitte s und halber Länge t genau dann ein Intervall mit Mitte m und halber Länge d schneidet, wenn gilt: $m - d \leq s + t$ und $s - t \leq m + d$

Abbildung 5.18 zeigt die Vorgehensweise bei der Bestimmung des nächstgelegenen Nachbarn (englisch: *nearest neighbor*). Suche zunächst auf dem Datenblock, der für den Query-Point Q zuständig ist, den nächstgelegenen Punkt P. Bilde eine *Range-Query* mit Quadrat um den Kreis um Q mit Radius $|P - Q|$. Schränke Quadratgröße weiter ein, falls nähere Punkte gefunden werden.

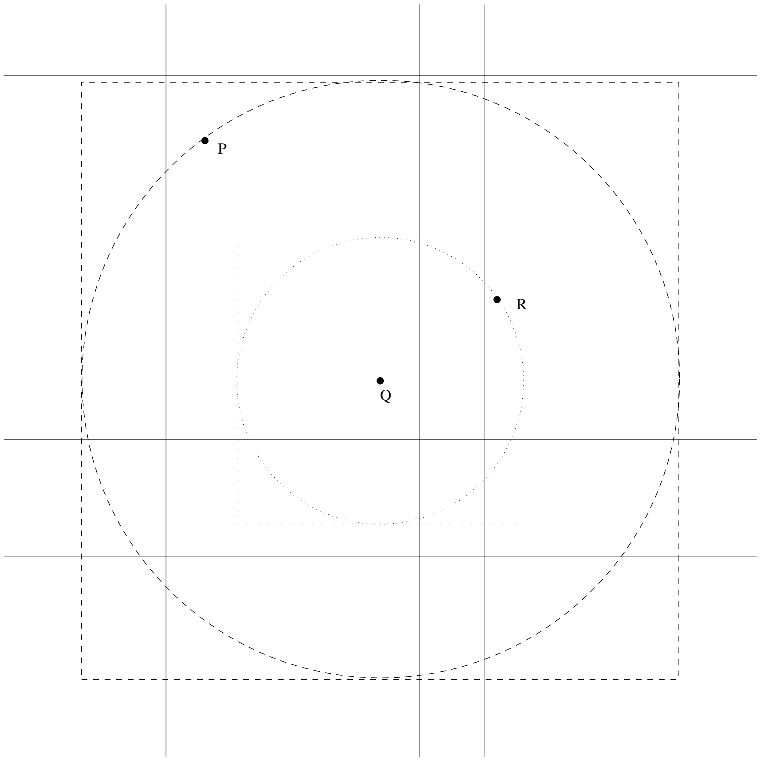

Abbildung 5.18: Nearest-Neighbor-Suche zu Query-Punkt Q

Die erwähnten Techniken lassen sich auf höherdimensionierte Geometrie-Objekte wie Rechtecke oder Quader erweitern. Zum Beispiel bietet sich zur Verwaltung von orthogonalen Rechtecken in der Ebene folgende Möglichkeit an: Ein Rechteck wird repräsentiert als ein Punkt im 4-dimensionalen Raum, gebildet durch die beiden 2-dimensionalen Punkte für horizontale bzw. vertikale Lage. Zu einem Query-Rechteck, bestehend aus horizontalem Intervall P und vertikalem Intervall Q, lassen sich die schneidenden Rechtecke finden im Durchschnitt der beiden kegelförmigen Abfragebereiche zu den Intervallen P und Q.