Kapitel 3

Logische Datenmodelle

In Abhängigkeit von dem zu verwendenden Datenbanksystem wählt man zur computergerechten Umsetzung des Entity-Relationship-Modells das hierarchische, das netzwerkorientierte, das relationale oder das objektorientierte Datenmodell.

3.1 Das Hierarchische Datenmodell

Datenbanksysteme, die auf dem hierarchischen Datenmodell basieren, haben (nach heutigen Standards) nur eine eingeschränkte Modellierfähigkeit und verlangen vom Anwender Kenntnisse der interne Ebene. Trotzdem sind sie noch sehr verbreitet (z.B. IMS von IBM), da sie sich aus Dateiverwaltungssystemen für die konventionelle Datenverarbeitung entwickelt haben. Die zugrunde liegende Speicherstruktur waren Magnetbänder, welche nur sequentiellen Zugriff erlaubten.

Im Hierarchischen Datenmodell können nur baumartige Beziehungen modelliert werden. Eine Hierarchie besteht aus einem Wurzel-Entity-Typ, dem beliebig viele Entity-Typen unmittelbar untergeordnet sind; jedem dieser Entity-Typen können wiederum Entity-Typen untergeordnet sein usw. Alle Entity-Typen eines Baumes sind verschieden.

Abbildung 3.1 zeigt ein hierarchisches Schema sowie eine mögliche Ausprägung anhand der bereits bekannten Universitätswelt. Der konstruierte Baum ordnet jedem Studenten alle Vorlesungen zu, die er besucht, sowie alle Professoren, bei denen er geprüft wird. In dem gezeigten Baum ließen sich weitere Teilbäume unterhalb der *Vorlesung* einhängen, z.B. die Räumlichkeiten, in denen Vorlesungen stattfinden. Obacht: es wird keine Beziehung zwischen den Vorlesungen und Dozenten hergestellt! Die Dozenten sind den Studenten ausschließlich in ihrer Eigenschaft als Prüfer zugeordnet.

Grundsätzlich sind einer Vater-Ausprägung (z.B. `Erika Mustermann`) für jeden ihrer Sohn-Typen jeweils mehrere Sohnausprägungen zugeordnet (z.B. könnte der Sohn-Typ `Vorlesung` 5 konkrete Vorlesungen enthalten). Dadurch entsprechen dem Baum auf Typ-Ebene mehrere Bäume auf Entity-Ebene. Diese Entities sind in Preorder-Reihenfolge zu erreichen, d.h. vom Vater zunächst seine Söhne und Enkel und dann dessen Brüder. Dieser Baumdurchlauf ist die einzige Operation auf einer Hierarchie; jedes Datum kann daher nur über den Einstiegspunkt Wurzel und von dort durch Überlesen nichtrelevanter Datensätze gemäß der

Preorder-Reihenfolge erreicht werden.

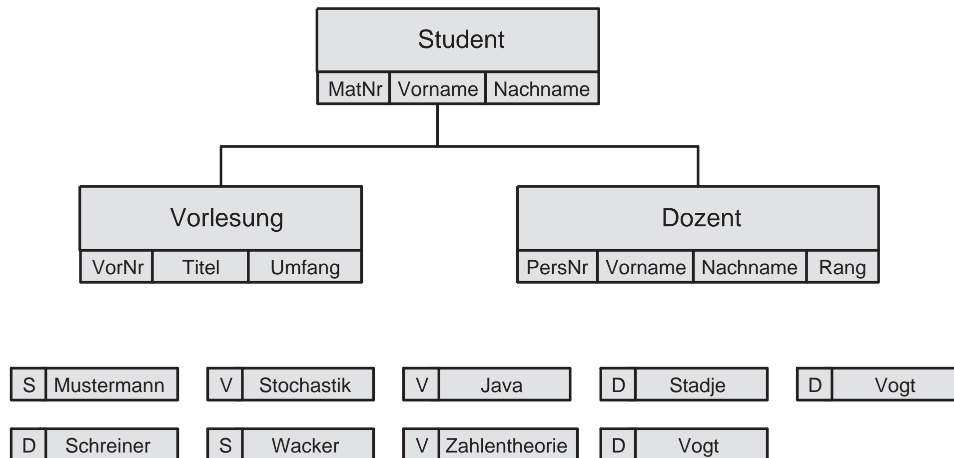

Abbildung 3.1: Hierarchisches Schema und eine Ausprägung.
Der Recordtyp sei erkennbar an den Zeichen S (Studenten), V (Vorlesungen) und D (Dozenten)

Die typische Operation besteht aus dem Traversieren der Hierarchie unter Berücksichtigung der jeweiligen Vaterschaft, d. h. der Befehl

```
GET NEXT Vorlesung WITHIN PARENT
```

durchläuft sequentiell ab der aktuellen Position die Preorder-Sequenz solange vorwärts, bis ein dem aktuellen Vater zugeordneter Datensatz vom Typ *Vorlesung* gefunden wird.

Beispiel-Query: Welche Hörer sitzen in der Vorlesung Zahlentheorie ?

1.) Einstieg in Baum mit Wurzel *Studenten*

2.) suche nächsten Student

3.) suche unter seinen Söhnen eine Vorlesung mit Titel `Zahlentheorie`

4.) falls gefunden: gib den Studenten aus

5.) gehe nach 2.)

Um zu vermeiden, daß alle Angaben zu den Dozenten mehrfach gespeichert werden, kann eine eigene Hierarchie für die Dozenten angelegt und in diese dann aus dem Studentenbaum heraus verwiesen werden.

3.2 Das Netzwerk-Datenmodell

Im Netzwerk-Datenmodell können nur binäre many-one- (bzw. one-many)-Beziehungen dargestellt werden. Ein E-R-Diagramm mit dieser Einschränkung heißt *Netzwerk*. Zur Formulierung der many-one-Beziehungen gibt es sogenannte *Set-Typen*, die zwei Entity-Typen in Beziehung setzen. Ein Entity-Typ übernimmt mittels eines Set-Typs die Rolle des *owner* bzgl. eines weiteren Entity-Typs, genannt *member*.

Im Netzwerk werden die Beziehungen als gerichtete Kanten gezeichnet vom Rechteck für *member* zum Rechteck für *owner* (funktionale Abhängigkeit). In einer Ausprägung führt ein gerichteter Ring von einer *owner*-Ausprägung über alle seine *member*-Ausprägungen (Abbildung 3.2).

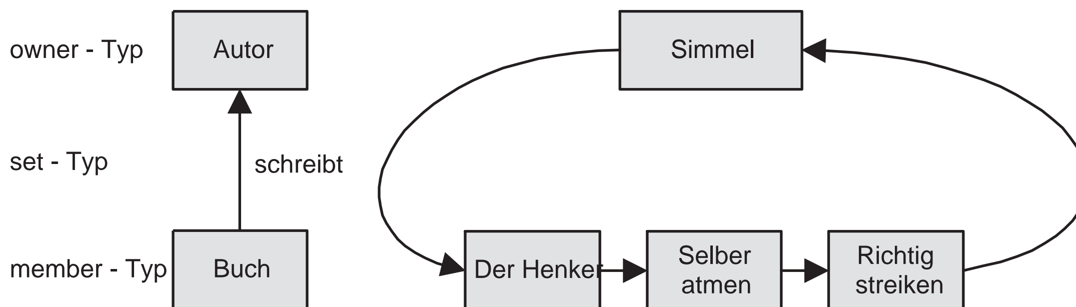

Abbildung 3.2: Netzwerkschema und eine Ausprägung

Bei nicht binären Beziehungen oder nicht many-one-Beziehungen hilft man sich durch Einführung von künstlichen *Kett-Records*. Abbildung 3.3 zeigt ein entsprechendes Netzwerkschema und eine Ausprägung, bei der zwei Studenten jeweils zwei Vorlesungen hören.

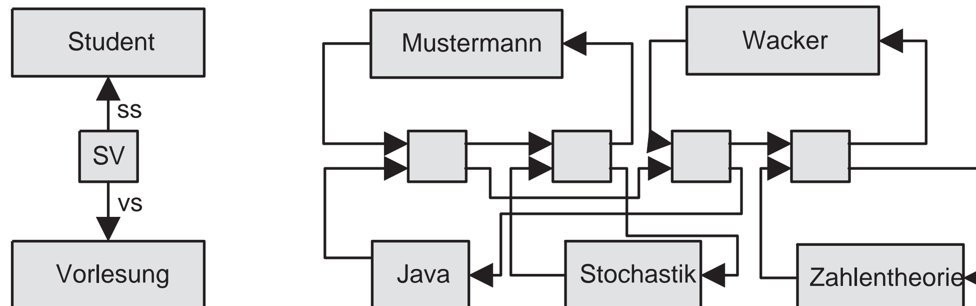

Abbildung 3.3: Netzwerkschema mit Kett-Record und eine Ausprägung

Die typische Operation auf einem Netzwerk besteht in der Navigation durch die verzeigerten Entities. Mit den Befehlen

```
FIND NEXT Student
FIND NEXT sv WITHIN ss
FIND OWNER WITHIN vs
```

lassen sich für einen konkreten Studenten alle seine Kett-Records vom Typ `sv` durchlaufen und dann jeweils der *owner* bzgl. des Sets `vs` ermitteln.

3.3 Das Relationale Datenmodell

Seien $D_1, D_2, \ldots, D_k$ Wertebereiche. $R \subseteq D_1 \times D_2 \times \ldots \times D_k$ heißt Relation. Wir stellen uns eine Relation als Tabelle vor, in der jede Zeile einem Tupel entspricht und jede Spalte einem bestimmten Wertebereich. Die Folge der Spaltenidentifizierungen heißt *Relationenschema*. Eine Menge von Relationenschemata heißt *relationales Datenbankschema*, die aktuellen Werte der einzelnen Relationen ergeben eine Ausprägung der relationalen Datenbank.

- pro Entity-Typ gibt es ein Relationenschema mit Spalten benannt nach den Attributen.
- pro Relationshiptyp gibt es ein Relationenschema mit Spalten für die Schlüssel der beteiligten Entity-Typen und ggf. weitere Spalten.

Abbildung 3.4 zeigt ein Schema zum Vorlesungsbetrieb und eine Ausprägung.

Student

MatNr	Vorname	Nachname
653467	Erika	Mustermann
875462	Willi	Wacker
432788	Peter	Pan

Hoert

MatNr	VorNr
653467	6.712
875462	6.712
432788	6.712
875462	6.102

Vorlesung

VorNr	Titel	Umfang
6.718	Java	4
6.174	Stochastik	2
6.108	Zahlentheorie	4

Abbildung 3.4: Relationales Schema und eine Ausprägung

Die typischen Operationen auf einer relationaler Datenbank lauten:

- **Selektion:**
  Suche alle Tupel einer Relation mit gewissen Attributeigenschaften
- **Projektion:**
  filtere gewisse Spalten heraus
- **Verbund:**
  Finde Tupel in mehreren Relationen, die bzgl. gewisser Spalten übereinstimmen.

Beispiel-Query: Welche Studenten hören die Vorlesung Zahlentheorie ?

```
SELECT Student.Nachname from Student, Hoert, Vorlesung
WHERE  Student.MatNr   = Hoert.MatNr
AND    Hoert.VorNr     = Vorlesung.VorNr
AND    Vorlesung.Titel = "Zahlentheorie"
```

3.4 Das Objektorientierte Datenmodell

Eine Klasse repräsentiert einen Entity-Typ zusammen mit darauf erlaubten Operationen. Attribute müssen nicht atomar sein, sondern bestehen ggf. aus Tupeln, Listen und Mengen. Die Struktur einer Klasse kann an eine Unterklasse vererbt werden. Binäre Beziehungen können durch mengenwertige Attribute modelliert werden.

Die Definition des Entity-Typen *Person* mit seiner Spezialisierung *Student* incl. der Beziehung *hoert* sieht im objektorientierten Datenbanksystem O_2 wie folgt aus:

```
class Person
  type tuple (name        : String,
              geb_datum   : Date,
              kinder      : list(Person))
end;

class Student inherit Person
  type tuple (mat_nr      : Integer,
              hoert       : set (Vorlesung))
end;

class Vorlesung
  type tuple (titel       : String,
              gehoert_von : set (Student))
end;
```

Kapitel 4

Physikalische Datenorganisation

4.1 Grundlagen

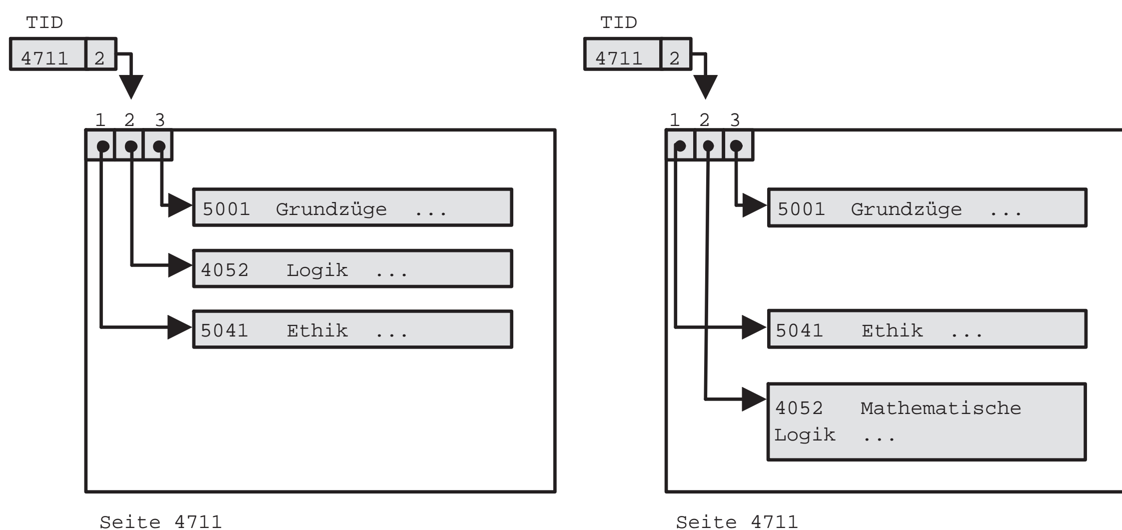

Abbildung 4.1: Verschieben eines Tupels innerhalb einer Seite

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.

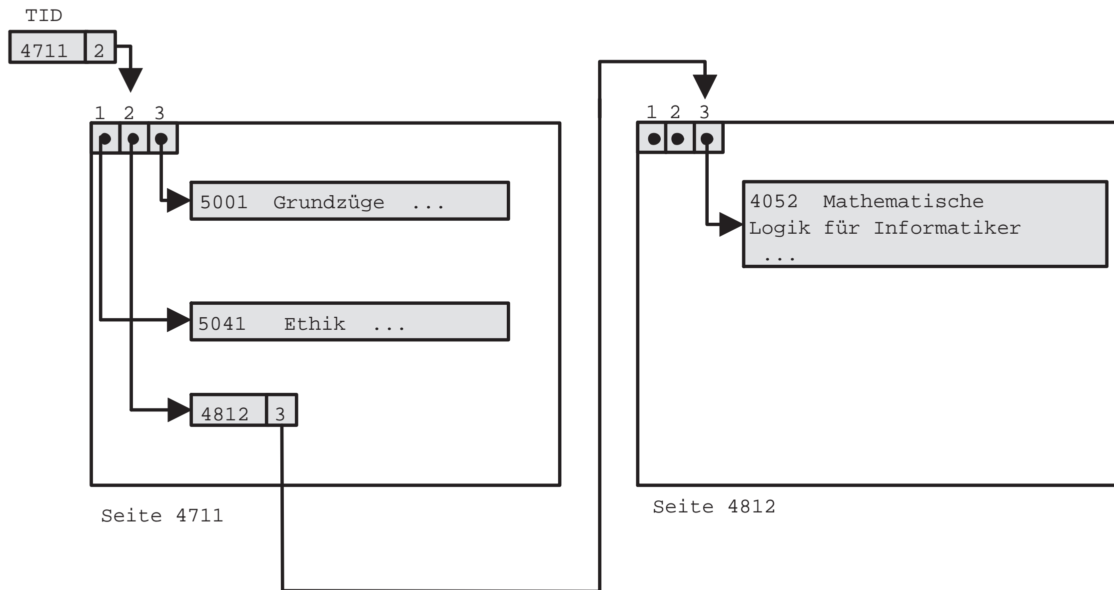

Abbildung 4.2: Verdrängen eines Tupels von einer Seite

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

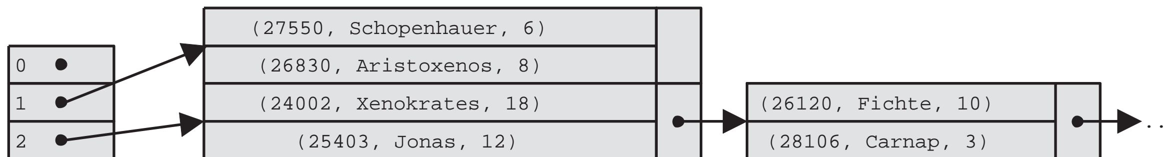

Abbildung 4.3: Hash-Tabelle mit Einstieg in Behälter

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 Anzahl 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

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

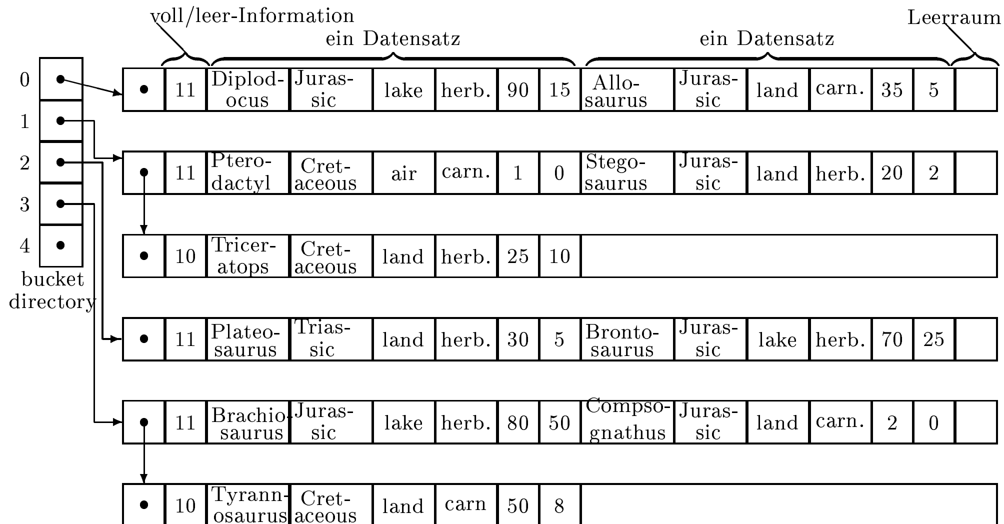

Abbildung 4.4: Hash-Organisation vor Einfügen von Elasmosaurus

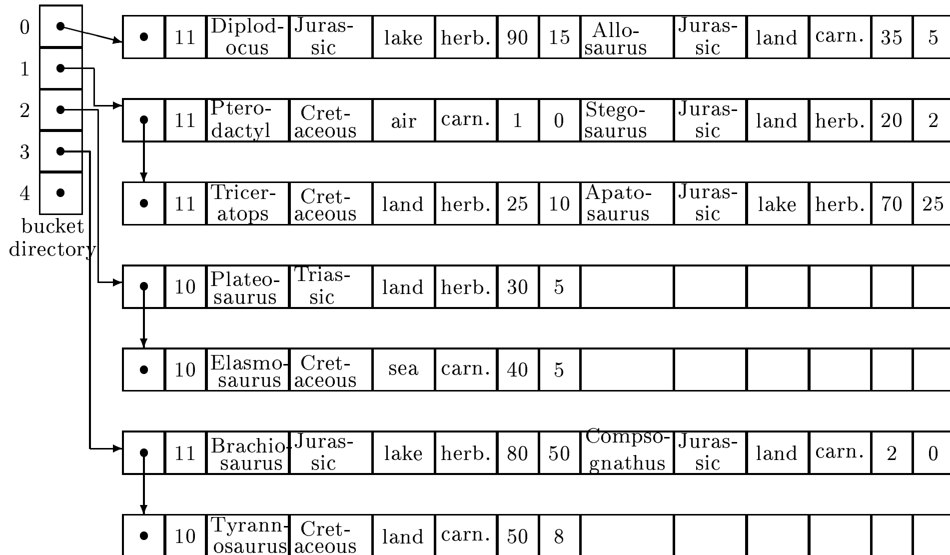

Abbildung 4.5: Hash-Organisation nach Einfügen von Elasmosaurus und Umbenennen

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 2 Bits pro Subblock

(Platz für ein Record), die angeben, ob dieser Subblock leer (also beschreibbar), gefüllt (also lesbar) oder gelöscht (also nicht zum Lesen geeignet) ist. Gelöschte Records werden wegen der Gefahr hängender Zeiger bis zum generellen Aufräumen nicht wieder verwendet.

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 beteiligt: DELETE und INSERT durchführen. Falls Schlüssel nicht beteiligt: 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 $v \bmod 5$. 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.

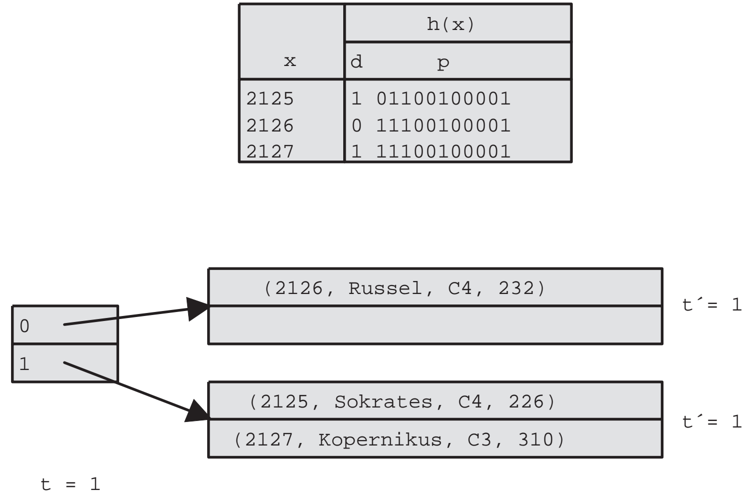

Abbildung 4.6: Hash-Index mit globaler Tiefe 1

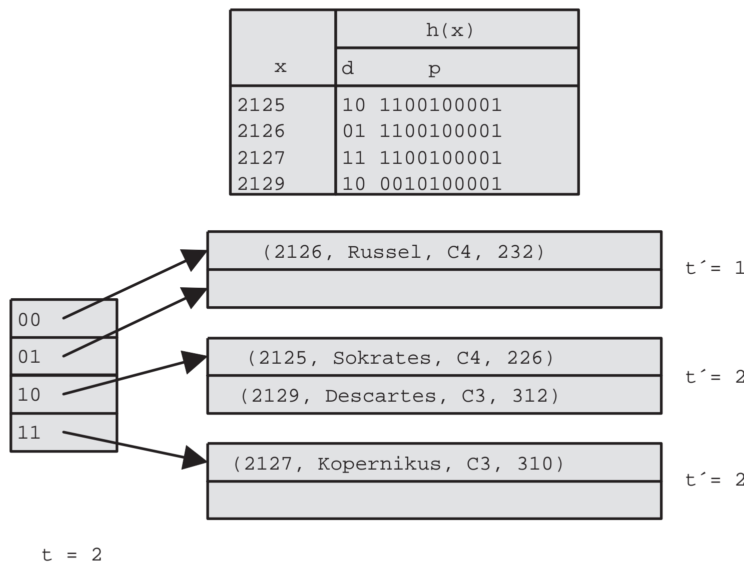

Abbildung 4.7: Hash-Index mit globaler Tiefe 2

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 Subblocks belegt sind. Dann ergeben sich die folgenden Operationen:

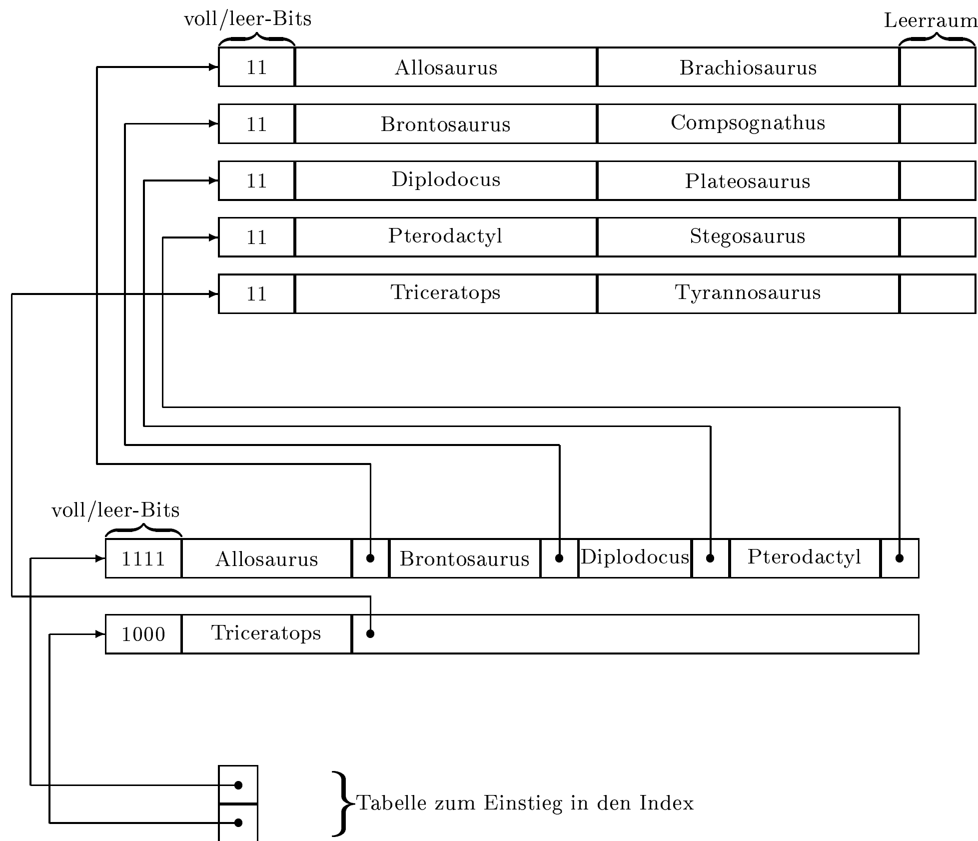

Abbildung 4.8: ISAM: Ausgangslage

- **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.

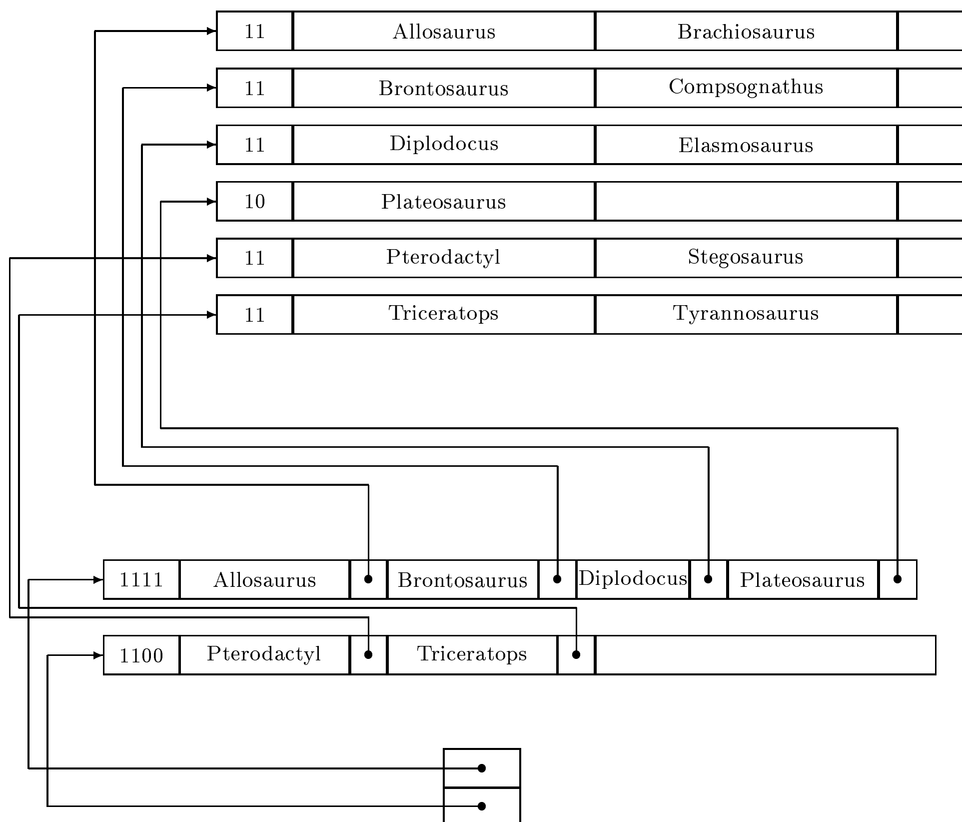

Abbildung 4.9: ISAM: nach Einfügen von Elasmosaurus

- **DELETE:** analog zu INSERT

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`.

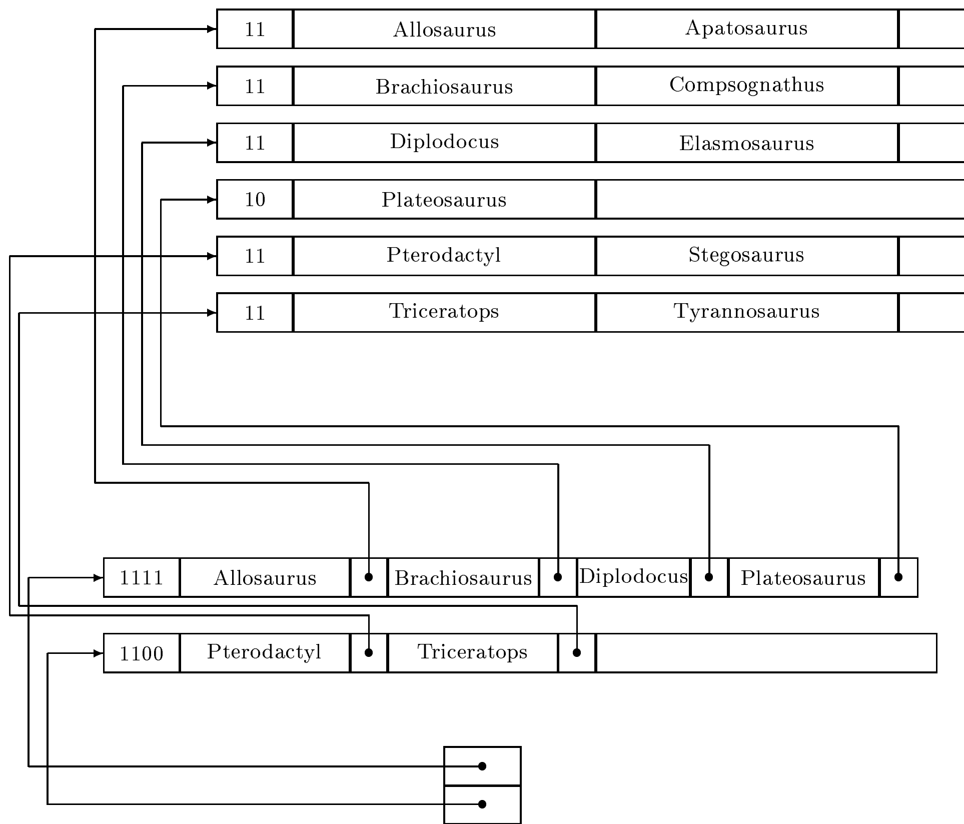

Abbildung 4.10: ISAM: nach Umbenennen von Brontosaurus

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.
```

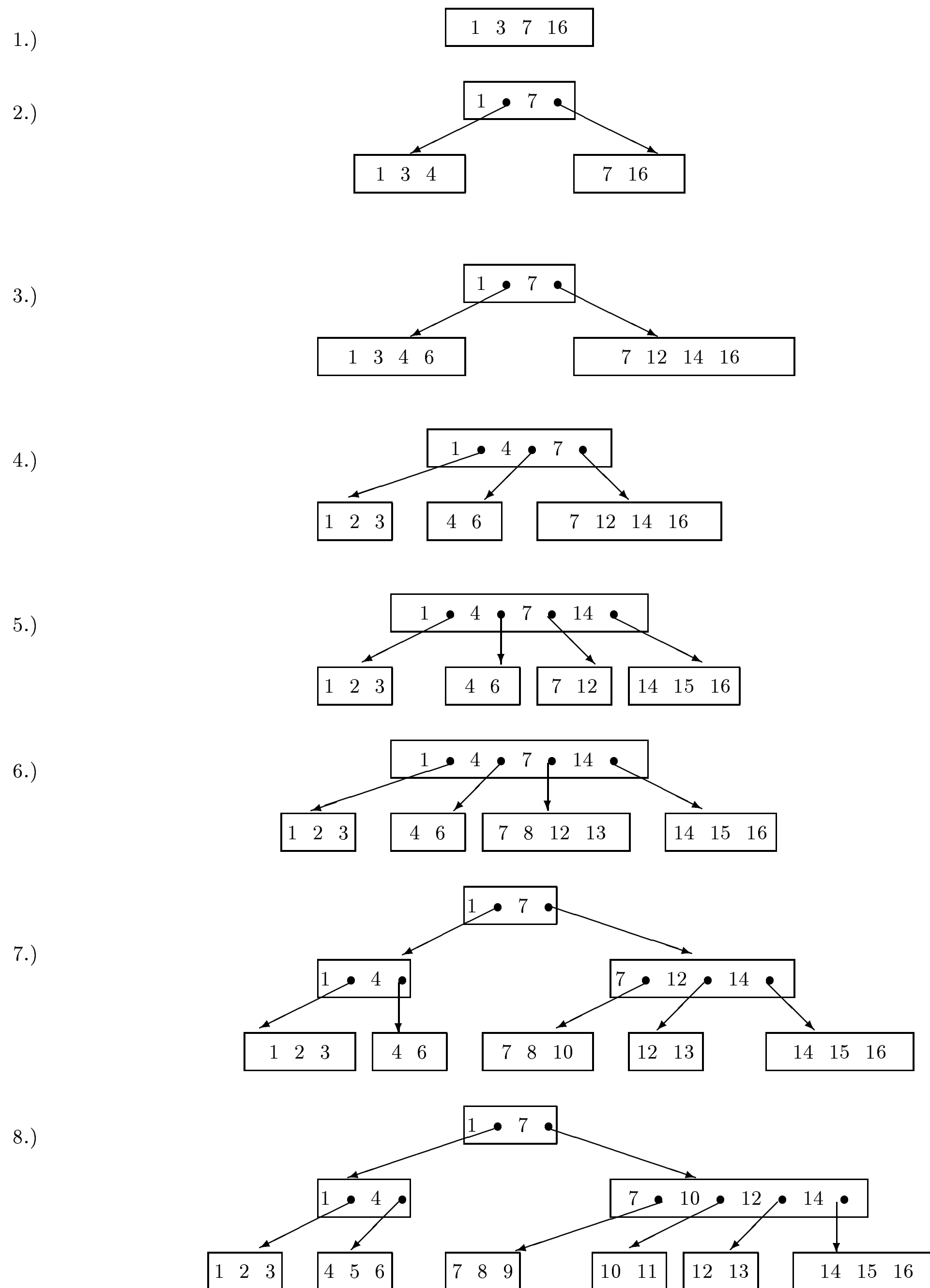

Abbildung 4.11: dynamisches Verhalten eines B*-Baums

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

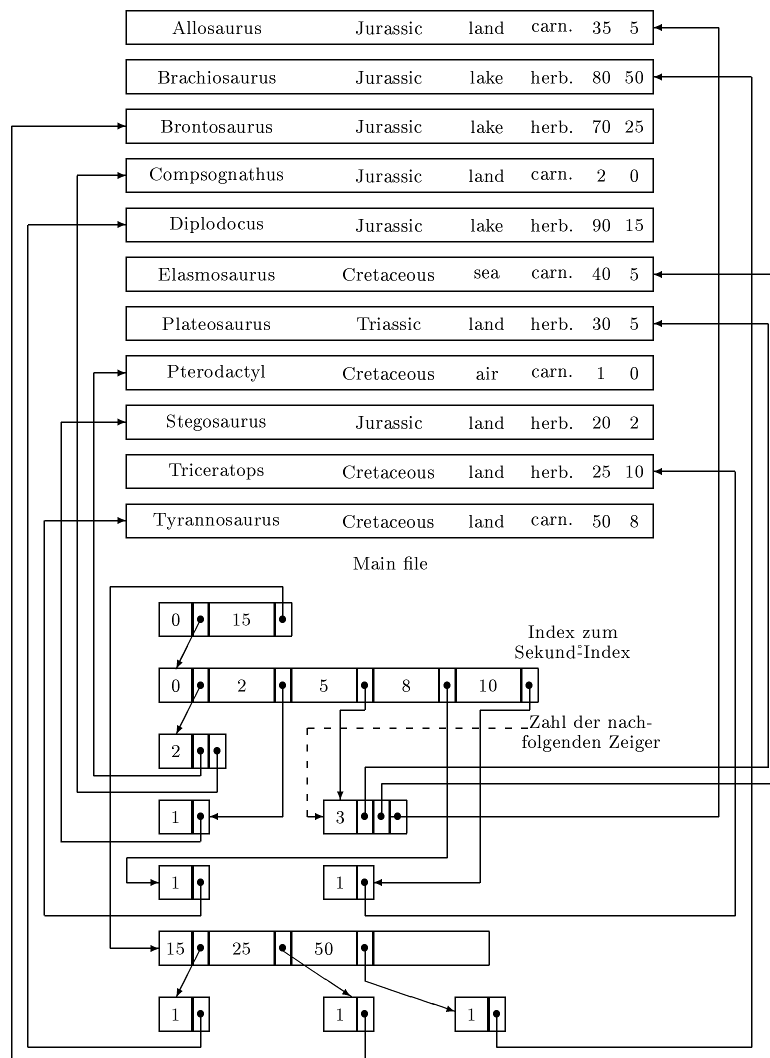

Abbildung 4.12: Sekundär-Index für GEWICHT

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.