Kapitel 4

Physikalische Datenorganisation

4.1 Grundlagen

Bedingt durch die unterschiedlichen Speichertechnologien weisen Hauptspeicher, Festplatte und Magnetband charakteristische Vor- und Nachteile auf. Folgende Tabelle zeigt die relativen Merkmale bezüglich Größe, Zugriffsgeschwindigkeit, Preis, Granularität und Dauerhaftigkeit. Im Vergleich zum direkt adressierbaren Hauptspeicher ist eine typische Festplatte etwa 1.000 mal größer, verursacht einen etwa 100.000 mal langsameren Zugriff und kostet nur ein Hunderstel bezogen auf ein gespeichertes Byte.

	Primärspeicher	Sekundärspeicher	Tertiärspeicher
Größe	klein	groß [10^3]	sehr gros
Tempo	schnell	langsam [10^{-5}]	sehr langsam
Preis	teuer	billig [10^{-2}]	billig
Granularität	fein	grob	grob
Dauerhaftigkeit	flüchtig	stabil	stabil

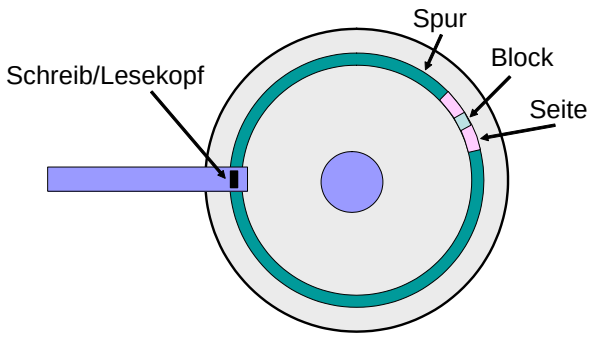

Abbildung 4.1: Schematischer Festplattenaufbau: Sicht von oben

Abbildung 4.1 zeigt den schematischen Aufbau einer Festplatte. Zum Lesen eines Blockes muss zunächst der Schreib-/Lesekopf oberhalb der zuständigen Spur positioniert werden (Seek Time), dann wird gewartet, bis der betreffende Block vorbeisaust (Latency Time), und schließlich kann der Block

übertragen werden (Transfer Time). Oft werden mehrere Blöcke nur zusammengefasst auf einer Seite übertragen.

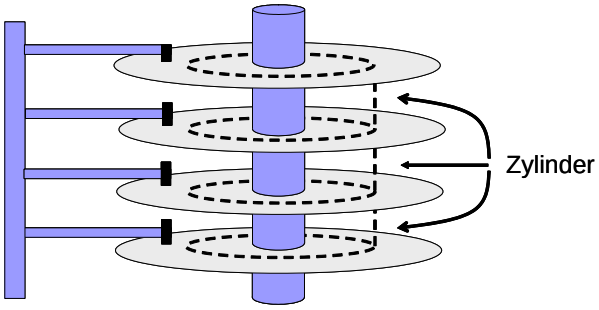

Abbildung 4.2: Schematischer Festplattenaufbau: Sicht von der Seite

Abildung 4.2 verdeutlicht, dass der Lesearm mehrere starr miteinander verbundene Schreib-/Leseköpfe gemeinsam bewegen und somit auf die jeweils übereinanderliegenden Spuren aller Magnetscheiben (genannt: Zylinder) gleichzeitig zugreifen kann. Der Block als kleinste direkt adressierbare Speichereinheit spielt daher für alle Datenstrukturen und Suchalgorithmen die zentrale Rolle.

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.

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 4.3 zeigt das Verschieben eines Datentupels innerhalb seiner ursprünglichen Seite; in Abbildung 4.4 wird das Record schließlich auf eine weitere Seite verdrängt.

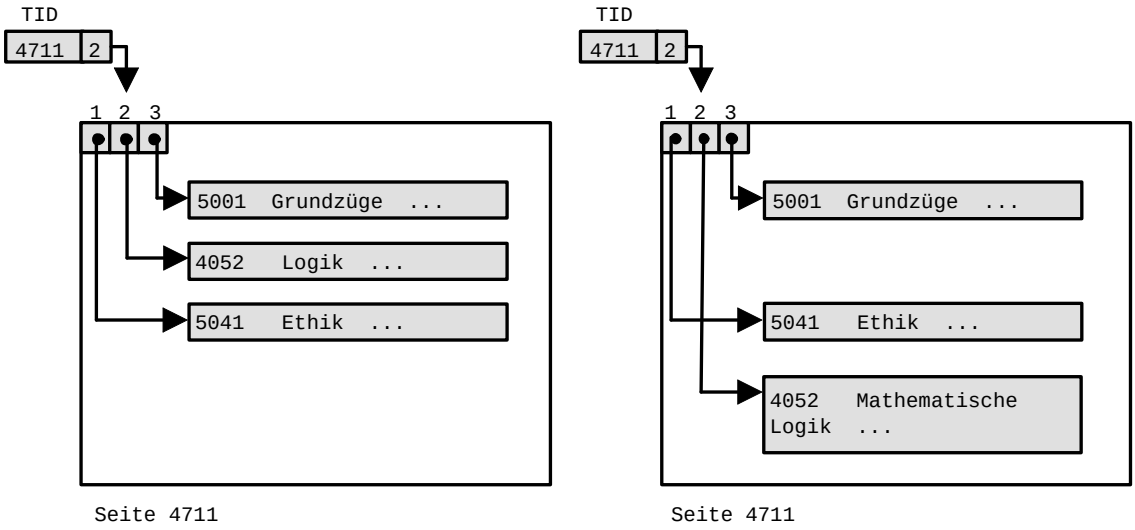

Abbildung 4.3: Verschieben eines Tupels innerhalb einer Seite

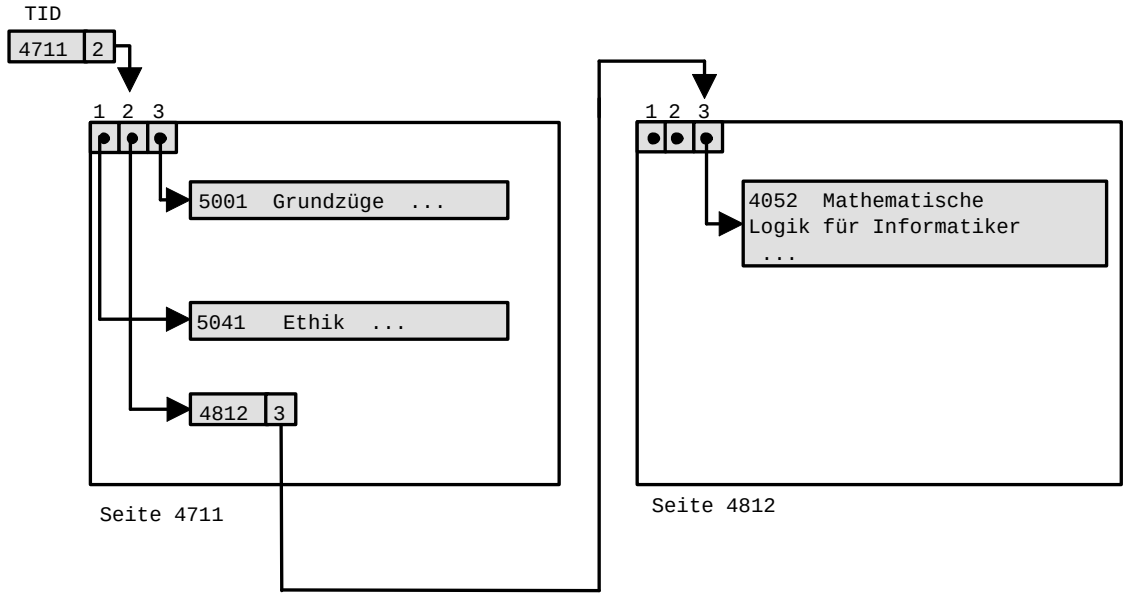

Abbildung 4.4: Verdrängen eines Tupels von einer Seite

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

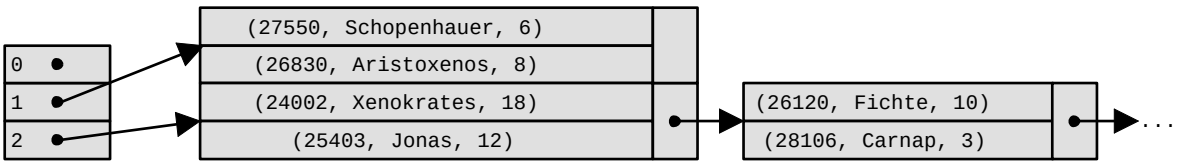

Abbildung 4.5: Hash-Tabelle mit Einstieg in Behälter

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

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

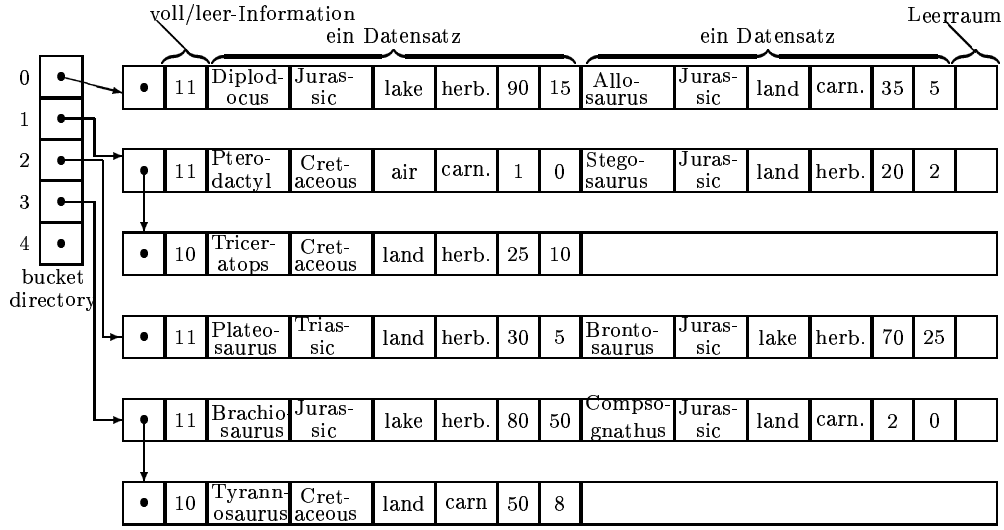

Abbildung 4.6: Hash-Organisation vor Einfügen von Elasmosaurus

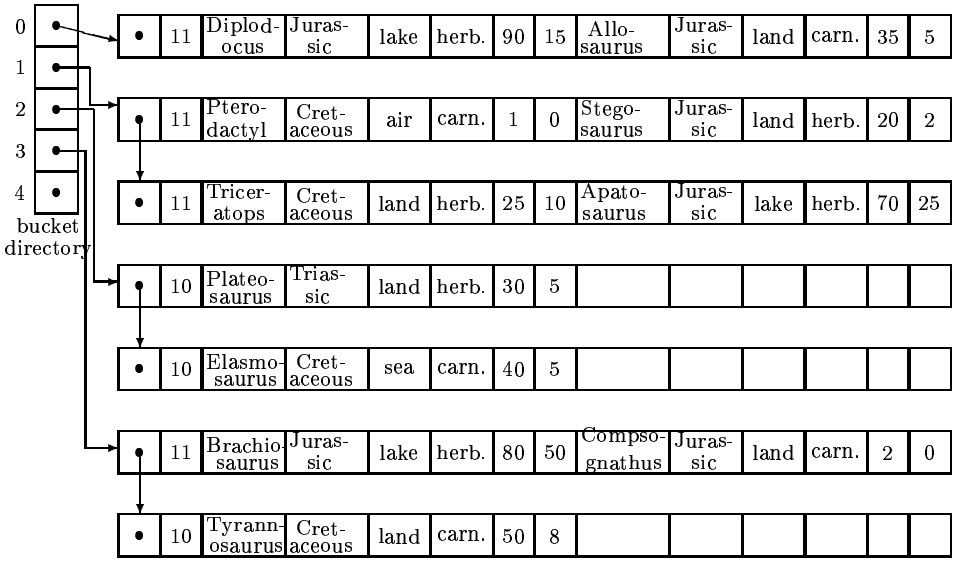

Abbildung 4.7: Hash-Organisation nach Einfügen von Elasmosaurus und Umbenennen

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 4.5 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 les-

bar) 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 4.6 und 4.7 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 4.6 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 4.7 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.

4.4 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*).

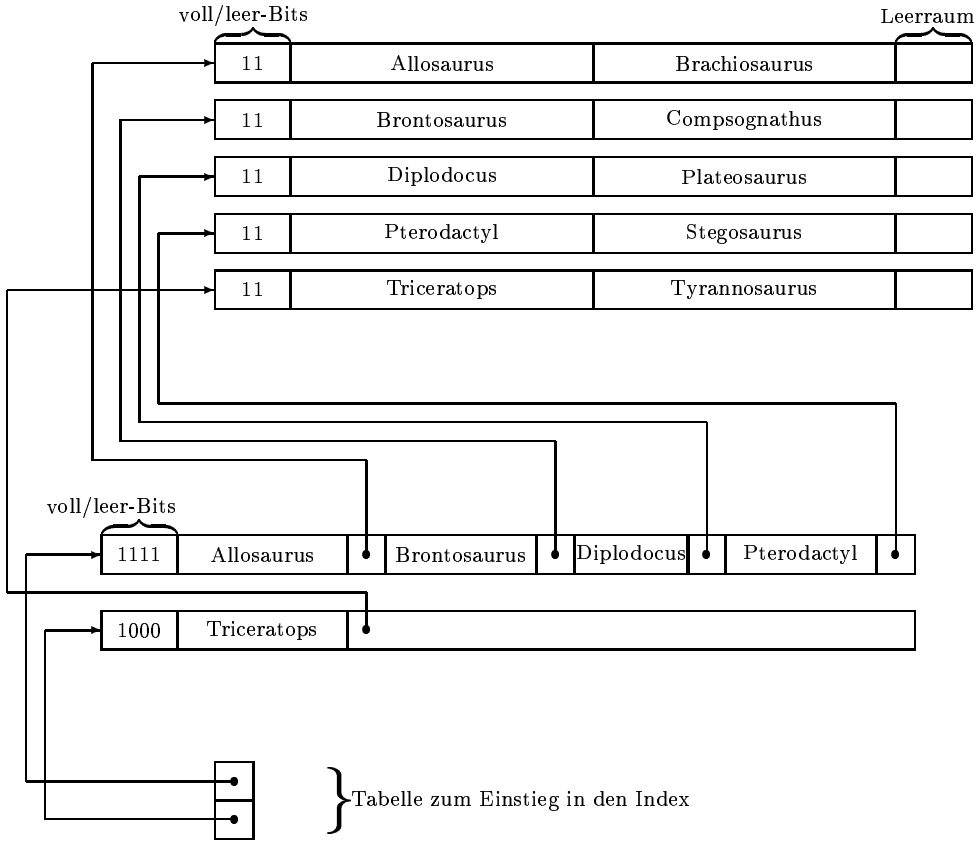

Abbildung 4.8: ISAM: Ausgangslage

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

- **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

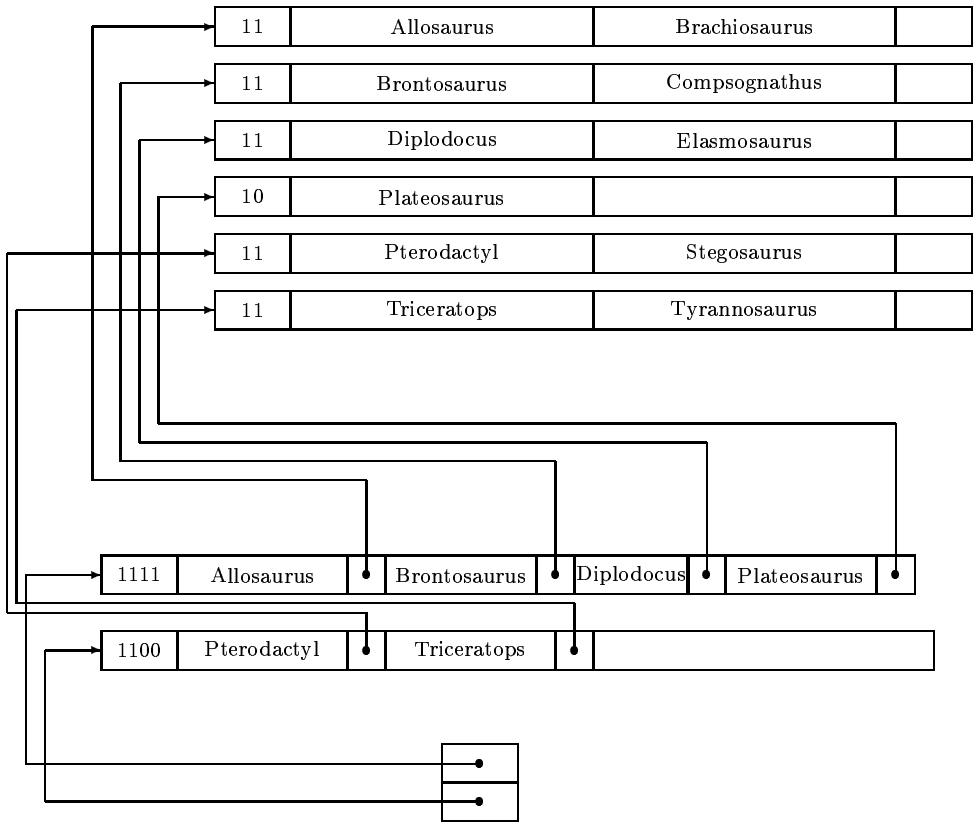

Abbildung 4.9: ISAM: nach Einfügen von Elasmosaurus

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.

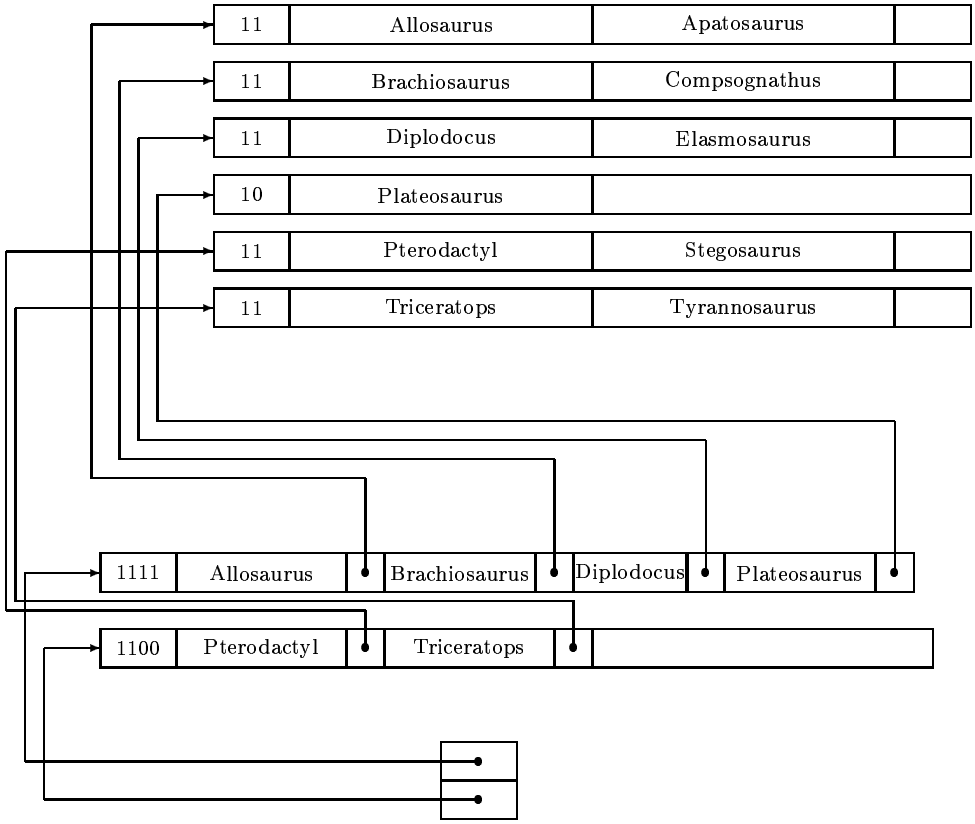

Abbildung 4.10: ISAM: nach Umbenennen von Brontosaurus

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).

4.5 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.

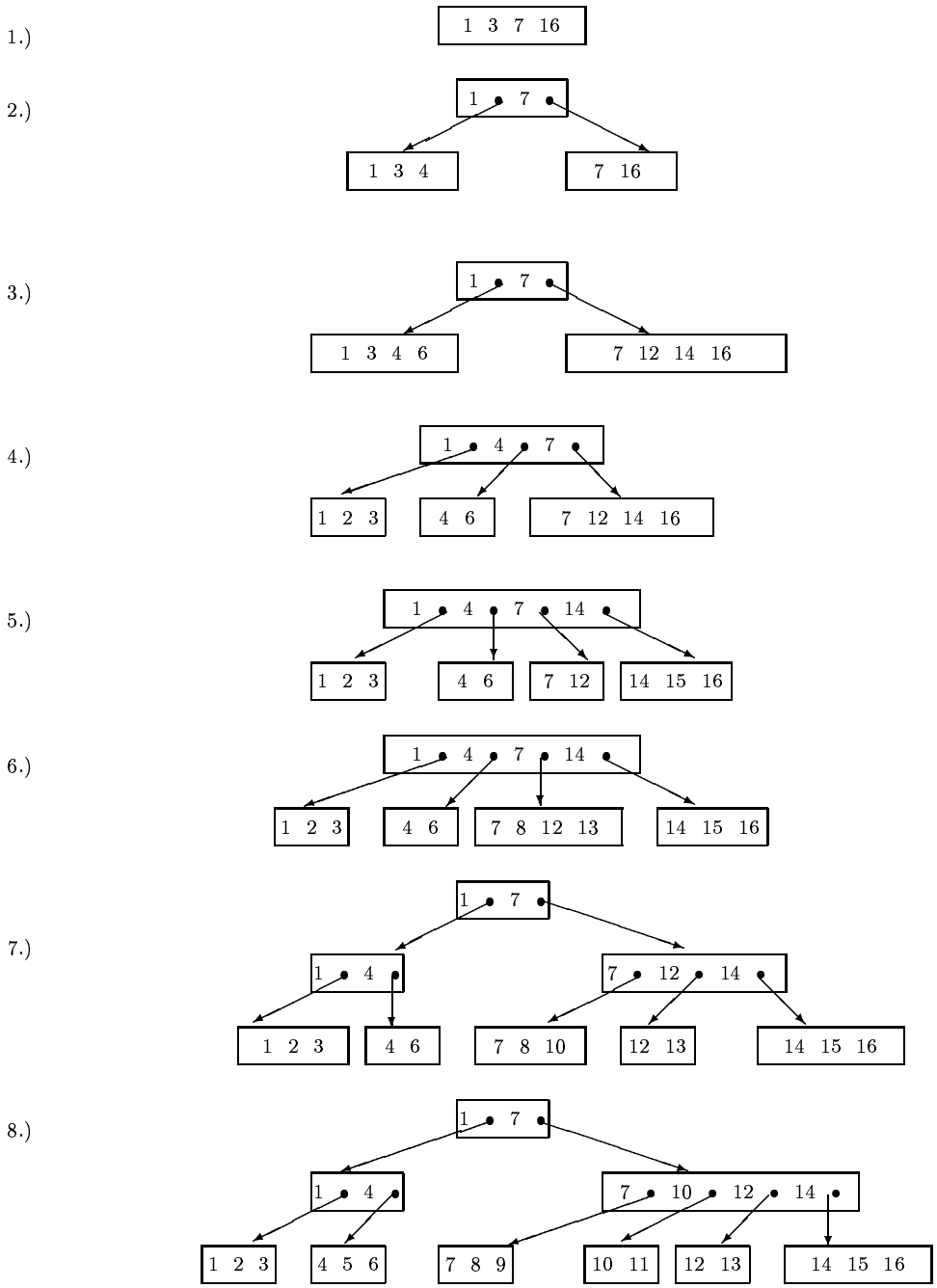

Abbildung 4.11: dynamisches Verhalten eines B*-Baums

Abbildung 4.11 zeigt das dynamische Verhalten eines B*-Baums mit dem Parameter $k = 2$. Es werden 16 Schlüssel eingefügt und 8 Schnappschüsse gezeichnet:

Schlüssel: `3,7,1,16,4,14,12,6,2,15,13,8,10,5,11,9`

	↑ ↑		↑ ↑	↑	↑	↑	↑
Zeitpunkt:	1. 2.		3. 4.	5.	6.	7.	8.

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

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

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.

4.6 Sekundär-Index

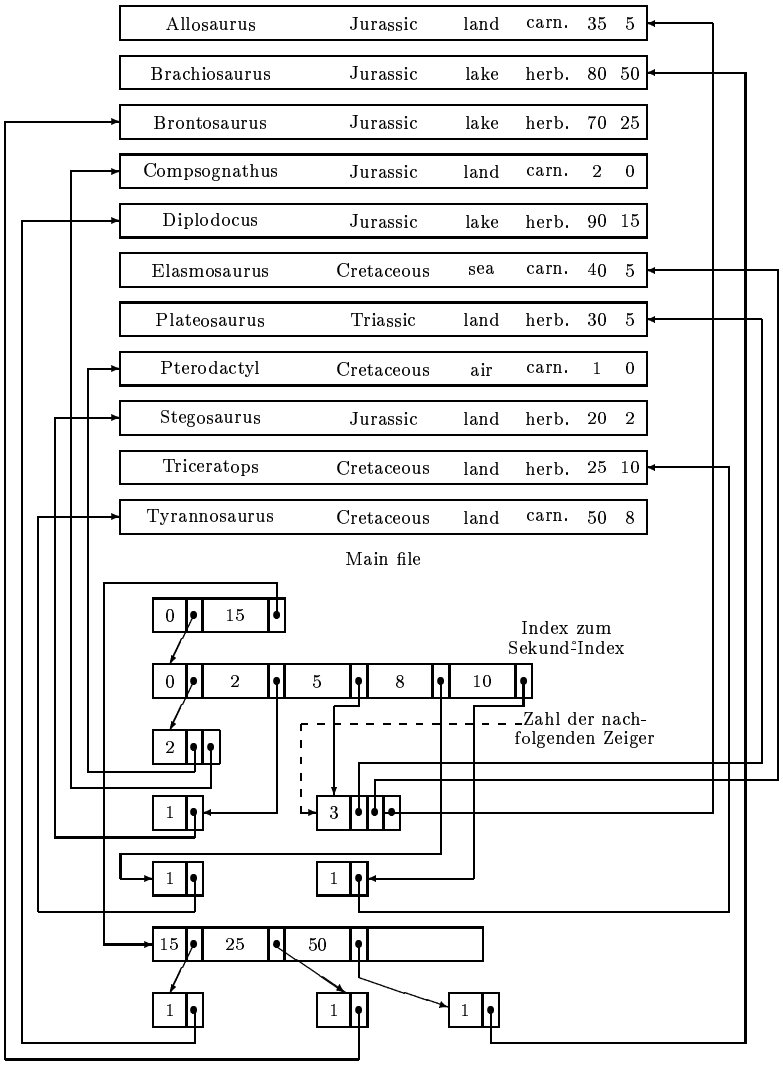

Abbildung 4.12: Sekundär-Index für GEWICHT

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.

4.7 Google

Als Beispiel für einen heftig genutzten Index soll die grundsätzliche Vorgehensweise bei der Firma *Google* besprochen werden.

Google wurde 1998 von Sergey Brin und Larry Page gegründet und ist inzwischen auf 2500 Mitarbeiter angewachsen. Die Oberfläche der Suchmaschine wird in mehr als 100 Sprachen angeboten. Etwa 8 Milliarden Webseiten liegen im Cache, der mehr als 40 TeraByte an Plattenplatz belegt. Im Lexikon befinden sich etwa 14 Millionen Schlüsselwörter, nach denen täglich mit mehr als 100 Millionen Queries gefragt wird.

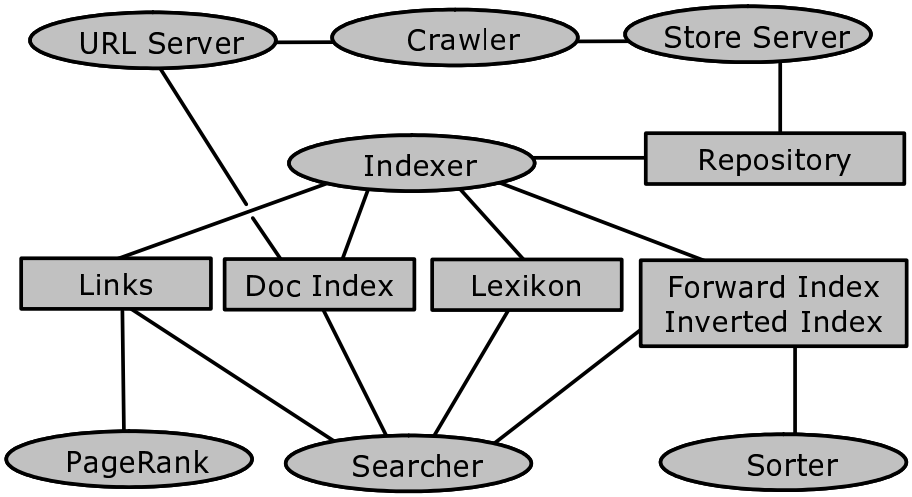

Abbildung 4.13: Google Architektur

Abbildung 4.13 zeigt die prinzipielle Systemarchitektur: Der *URL-Server* generiert eine Liste von URLs, welche vom *Crawler* systematisch besucht werden. Dabei werden neu entdeckte URLs im *DocIndex* abgelegt. Der *Store Server* legt den kompletten, abgerufenen HTML-Text eines besuchten Dokuments in komprimierter Form im *Repository* ab. Der *Indexer* verarbeitet den Repository-Inhalt: Jedem Dokument wird eine eindeutige `docID` zugeteilt. Aus jedem HREF-Konstrukt, welches von Seite A zur Seite B zeigt, wird ein Paar (docID(A), docID(B)) erzeugt und in der Liste aller Verweis-Paare eingefügt (*Links*). Jedes beobachtete Schlüsselwort wird ins *Lexikon* eingetragen zusammen mit einer eindeutigen *wordID*, der Zahl der relevanten Dokumente und einem Verweis auf die erste Seite des *Inverted Index*, auf der solche `docID`s liegen, in dessen Dokumenten dieses Wort beobachtet wurde oder auf die es innerhalb eines Anchor-Textes verweist.

Die Zeilen im Lexikon haben daher die Struktur

```
wordID #docs pointer
wordID #docs pointer
wordID #docs pointer
wordID #docs pointer
```

Durch Inspektion der Webseiten im Repository erstellt der *Indexer* zunächst den *Forward Index*, welcher zu jedem Dokument und jedem Wort die Zahl der Hits und ihre Positionen notiert.

Die Struktur im Forward Index lautet

```
docID wordID #hits hit hit hit hit hit hit
      wordID #hits hit hit hit
docID wordID #hits hit hit hit hit
      wordID #hits hit hit hit hit hit
      wordID #hits hit hit hit hit
```

Daraus erzeugt der *Sorter* den *Inverted Index*, welcher folgende Struktur aufweist:

```
wordID docID #hits hit hit hit hit hit hit
       docID #hits hit hit hit
       docID #hits hit hit hit hit hit
wordID docID #hits hit hit hit hit
       docID #hits hit hit hit hit hit
```

Aus technischen Gründen ist der Forward Index bereits auf 64 *Barrels* verteilt, die jeweils für einen bestimmten WordID-Bereich zuständig sind. Hierdurch entsteht beim Einsortieren zwar ein Speicheroverhead, da die Treffer zu einer docID sich über mehrere Barrels verteilen, aber die anschließende Sortierphase bezieht sich jeweils auf ein Barrel und kann daher parallelisiert werden.

Jeder *hit* umfasst zwei Bytes: 1 Capitalization Bit, 3 Bits für die Größe des verwendeten Fonts und 12 Bit für die Adresse im Dokument. Dabei werden alle Positionen größer als 4095 auf diesen Maximalwert gesetzt. Es wird unterschieden zwischen *plain hits*, welche innerhalb von normalem HTML auftauchen, und *fancy hits*, welche innerhalb einer URL, eines HTML-Title, im Anchor-Text oder in einem Meta-Tag auftauchen.

Der PageRank-Algorithmus berechnet auf Grundlage der Seitenpaare in der Datei *Links* die sogenannte *Link Popularität*, welche ein Maß für die Wichtigkeit einer Seite darstellt: Eine Seite ist wichtig, wenn andere wichtige Seiten auf sie verweisen.

Zur formalen Berechnung machen wir folgende Annahmen: Seite T habe $C(T)$ ausgehende Links. Auf Seite A mögen die Seiten $T_1, T_2, ..., T_n$ verweisen. Gegeben sei ein Dämpfungsfaktor $0 \leq d \leq 1$. (1-d) modelliert die Wahrscheinlichkeit, dass ein Surfer im Web eine Seite nicht über einen Link findet, sondern eine Verweiskette schließlich abbricht und die Seite spontan anwählt.

Dann ergibt sich der *PageRank* von Seite A wie folgt:

$$PR(A) := (1-d) \; + \; d \cdot \sum_{i=1}^{n} \frac{PR(T_i)}{C(T_i)}$$

Je wichtiger Seite T_i ist (großes $PR(T_i)$) und je weniger ausgehende Links sie hat (kleines $C(T_i)$), desto mehr strahlt von ihrem Glanz etwas auf Seite A.

Hierdurch entsteht ein Gleichungssystem, welches sich durch ein Iterationsverfahren lösen lässt; Google braucht dafür ein paar Stunden.

Abbildung 4.14 zeigt ein Mini-Web mit vier untereinander verzeigerten Seiten A, B, C, D zusammen mit den ermittelten PageRanks, basierend auf einem Dämpfungsfaktor von $d = 0.85$.

Der vorberechnete PageRank und der vorberechnete Inverted Index erlauben nun eine effiziente Suche nach einem oder mehreren Schlüsselwörtern.

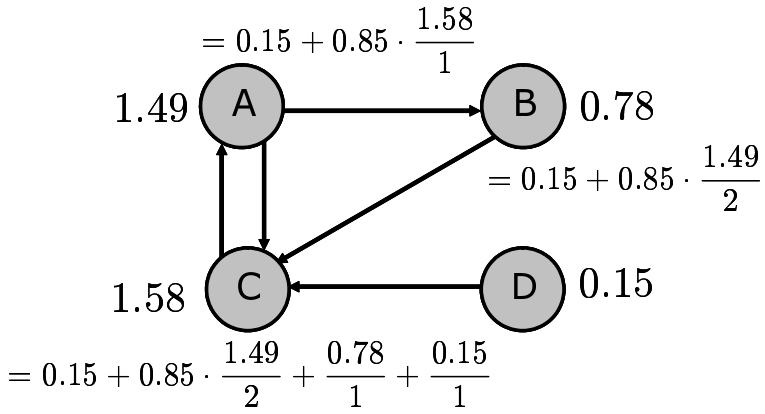

Abbildung 4.14: PageRank errechnet mit Dämpungsfaktor d=0.85

Bei einer Single-Word-Query w werden zunächst mit Hilfe des Inverted Index alle Seiten ermittelt, in denen w vorkommt. Für jede ermittelte Seite d werden die Hit-Listen ausgewertet bzgl. des Treffer-Typs (abnehmende Wichtigkeit für Treffer in title, anchor, URL, plain text large font, plain text small font, ...). Der gewichtete Treffervektor wird skalar multipliziert mit dem Trefferhäufigkeitsvektor und ergibt den *Weight-Score(d,w)*. Dieser wird nach einem geheimen Google-Rezept mit dem *PageRank(d)* kombiniert und es entsteht der *Final-Score(d,w)*, welcher durch Sortieren die Trefferliste bestimmt.

Bei einer Multi-Word-Query $w_1, w_2, ..., w_n$ werden zunächst 10 Entfernungsklassen gebildet (von 'unmittelbar benachbart' bis 'sehr weit entfernt') und als Grundlage für einen sogenannten *Proximity-Score*$(d, w_1, w_2, ..., w_n)$ ausgewertet: nah in einem Dokument beeinanderliegende Suchwörter werden belohnt. Dann wird für jede gemeinsame Trefferseite d und jedes Suchwort w_i der konventionelle *Weight-Score*(d, w_i) kombiniert mit dem *Proximity-Rank*$(d, w_1, w_2, ..., w_n)$ und dem *PageRank(d)*. Der dadurch errechnete *Final-Score*$(d, w_1, w_2, ..., w_n)$ bestimmt nach dem Sortieren die Trefferliste.