INDEXDATEIEN
(INDEXED FILES)

ISAM (Indexed Sequential Access Method)

- Sätze werden nach ihren Schlüsselwerten sortiert.
    - Schlüsselwerte sind **immer** vergleichbar und daher auch sortierbar. (Speicherung als Bit-Strings)

Ordnungen der Standard-Datentypen

Ganzzahlen, reelle Zahlen (Integer, Real):

→ numerische Ordnung.

Zeichenketten (Character Strings):

→ lexikographische (alphabetische) Ordnung.

Die **lexikographische Ordnung** wird definiert durch die Ordnung: $X_1 X_2 \cdots X_k < Y_1 Y_2 \cdots Y_m$

wobei X und Y jeweils **Zeichen** sind, wenn

1. $k < m$ und $X_1 X_2 \cdots X_k = Y_1 Y_2 \cdots Y_k$, oder wenn
2. Für ein $i \leq \min(k,m)$ gilt, daß $X_1 = Y_1, X_2 = Y_2, \ldots, X_{i-1} = Y_{i-1}$ und der numerische Code für X_i ist numerisch kleiner als der für Y_i.

Beispiele:

‘UN’ < ‘UND’

(Regel 1)

‘BETRAG’ < ‘BETRUG’

(Regel 2 mit i=5)

B = B

E = E

T = T

R = R

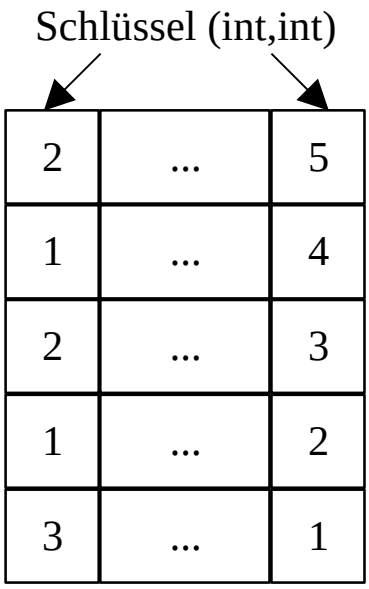
A < U

Schlüssel aus mehreren Feldern

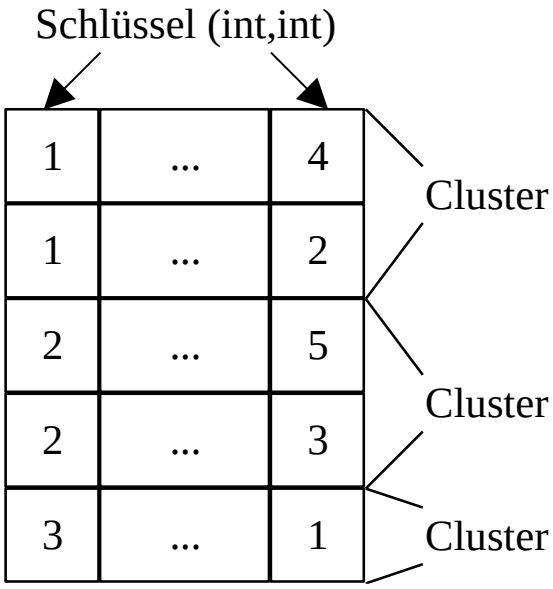
Wenn ein Schlüssel aus mehr als einem Feld besteht, werden die Sätze nach dem ersten Feld sortiert, wobei **Cluster** entstehen, in denen der Wert im ersten Feld gleich ist. Diese Cluster werden nach dem zweiten Feld sortiert ...

Dies stellt nur eine Generalisierung der lexikographischen Ordnung dar.

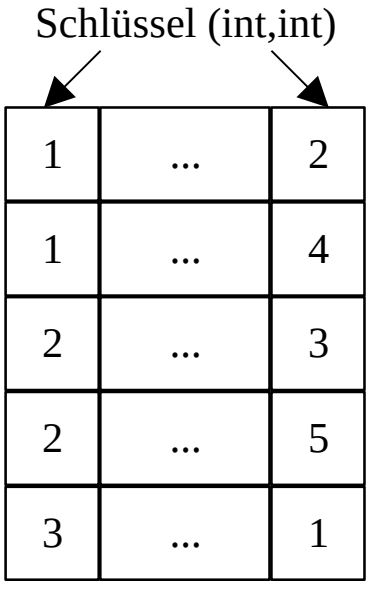
Beispiel:



Sortiert nach Feld 1:



Sortiert nach Feld 2:



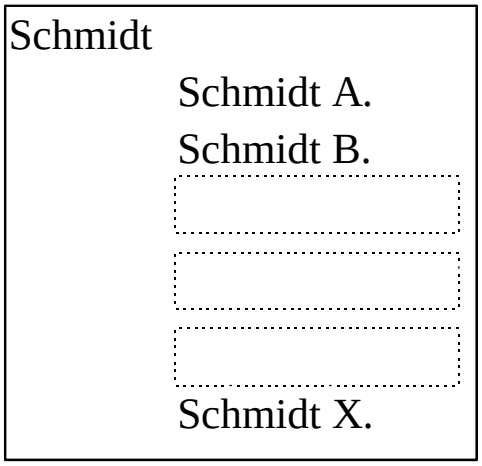
Kosten: Die Datei muß über den Schlüsselwerten sortiert sein und bleiben (Insert).

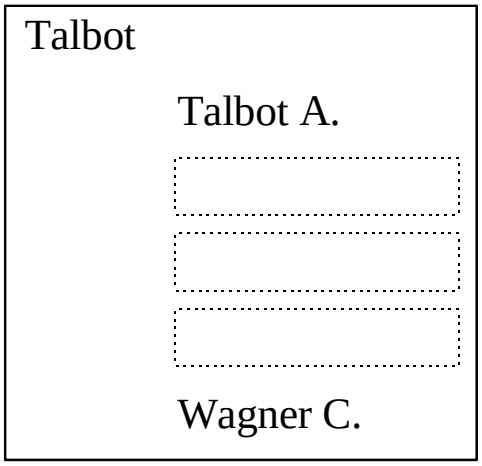
Vorteil: Die Operation Lookup wird sehr schnell ausgeführt (wenn der Schlüsselwert bekannt ist).

Beispiele:

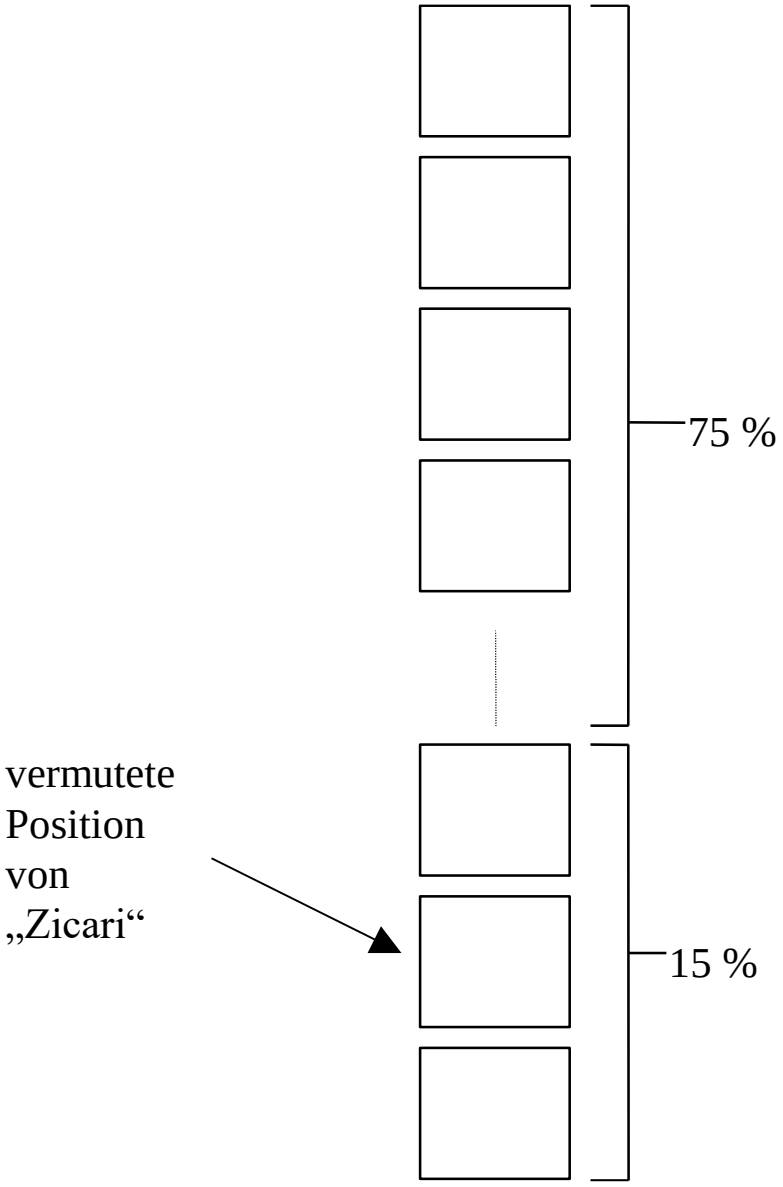
- Wörterbuch
- Telefonbuch

Sowohl bei Telefonbüchern als auch bei Wörterbüchern findet sich in der oberen Ecke das erste Wort (oder der erste Name) auf der Seite:





In der Praxis ermöglicht dies, die Seite mit dem richtigen Wort oder Namen durch das Wissen über die Verteilung der Worte zu „erraten“, zumindest aber einzugrenzen.



Indexdateien:

Definition: Ein Index I zu einer Datei D ist eine Datei, deren Sätze Paare der Form (v_i, b_i) darstellen, wobei v_i der Schlüsselwert eines Satzes in D ist und b_i die Adresse dieses Satzes. D wird als Hauptdatei (main file) bezeichnet.

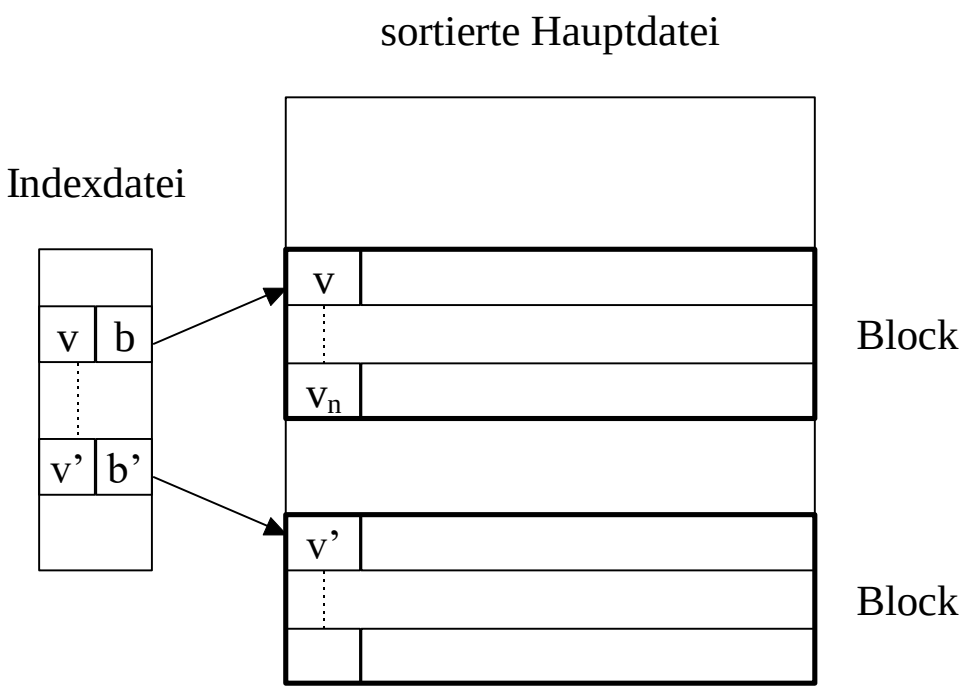
Annahme:

- Die Schlüsselwerte einer Datei sind geordnet.
- Der Index ist nach den Schlüsselwerten sortiert.
- Die Index Records sind unpinned.

Sparse Index

Hauptdatei sortiert gespeichert. Nur ein Indexeintrag pro Block der Hauptdatei; dieser Eintrag enthält den niedrigsten im Block gespeicherten Schlüssel.

Beispiel: (v_i,b_i) : b_i ist die Adresse des Blockes B_i, dessen erster Schlüsselwert v_i ist.



Suche nach einem Schlüssel *v* :

Durchsuche den Index sequentiell, bis ein Satz (v_1,b) gefunden ist, für den gilt:

1. $v_1 \leq v$ und
2. (v_1,b) ist der letzte Satz, oder für den Schlüssel v_2 des nächsten Satzes im Index gilt $v < v_2$.

Man sagt: „v_1 *überdeckt (covers)* v“.

Durchsuche den gefundenen Block nach dem gesuchten Satz.

Die Sortierreihenfolge der Hauptdatei muß innerhalb der Blöcke nicht eingehalten werden.

Suchen in einem Index

Die Aufgabe ist, zu einem gegebenen v einen Satz (v_1,b) im Index zu finden, so daß v von v_1 überdeckt wird.

1. lineare Suche:

Für einen Index mit n Blöcken müssen im Schnitt

$$\lceil n/2 \rceil \text{ Blockzugriffe}$$

erfolgen.

Nur für sehr kleine Indizes ausreichend schnell, aber immer noch besser als das Durchsuchen der ganzen Hauptdatei. Für eine Hauptdatei mit c Sätzen pro Block hat der Index nur 1/c-tel soviel Einträge wie die Hauptdatei. Außerdem passen mehr Index-Sätze in einen Block als Hauptdatei-Sätze.

2. binäre Suche:

Gegeben ist der Schlüssel *v* und ein Index der in den Blöcken $B_1, B_2, \ldots, B_n$ gespeichert ist.

Betrachtet wird dann der mittlere Block $B_{\lceil n/2 \rceil}$ und der Wert v_1 des ersten Satzes in diesem Block wird mit *v* verglichen.
Falls

$v < v_1$: weiter mit den Blöcken $B_1 \cdots B_{\lceil n/2 \rceil - 1}$.

$v \geq v_1$: weiter mit den Blöcken $B_{\lceil n/2 \rceil} \cdots B_n$.

Wenn nur noch ein Block übrig ist, wird dieser linear nach dem Schlüssel *v* durchsucht.

Bemerkung: Für die Abbildung der (errechneten) Werte *i* auf die Adresse des Blocks B_i, wird eine **Tabelle** benötigt.

Zeitverhalten:

Da die Anzahl der Blöcke mit jedem Schritt halbiert wird, ist die Suche nach höchstens $\log_2(n+1)$ Schritten beendet.

Es werden also ca. $\log_2 n$ Blöcke des Index in den Hauptspeicher geladen. Dann wird der Block der Hauptdatei geladen und evtl. Wieder geschrieben.

Insgesamt kommt die binäre Suche im Index auf

$\boxed{3+\log_2 n}$ Blockzugriffe.

Beispiel:

Hauptdatei:

- 1.000.000 Sätze.
- 10 Sätze in jedem Block.
  = 100.000 Blocks.

Index:

- 100.000 Sätze.
- 100 Sätze in jedem Block.
  = 1.000 Blocks

Lineare Suche:

$$\frac{1.000}{2} = 500 \text{ Blockzugriffe}$$

Binäre Suche:

$$3 + \log_2 1.000 \cong 13 \text{ Blockzugriffe}$$

Beispiel (Fortsetzung):

Hashing:

Im optimalen Fall (Gleichverteilung) benötigt Hashing zum finden eines Satzes nur

$$3$$ Blockzugriffe!

Dazu muss jeder Block der Hauptdatei mit 10 Sätzen gefüllt sein und die Bucketgröße auf 100.000 gesetzt sein.

Nachteile:

- Es ist schwierig, Sätze sortiert aufzulisten oder zu bearbeiten.
- Keine „Range Queries“.

Dense Index

Für jeden Satz der Hauptdatei ist im Index das entsprechende Schlüssel/Zeiger-Paar gespeichert und die Hauptdatei ist beliebig auf Blöcke verteilt (nicht sortiert!).

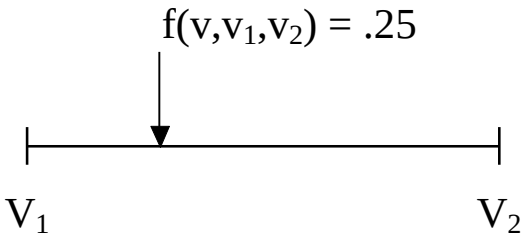
Vorteil: Sätze oder Index unpinned

1. Sätze unpinned: bessere Ausnutzung der Blöcke in der Hauptdatei.
2. Hauptdatei pinned, aber Index nicht, daher effizientere Zugriffsstruktur auf den Index möglich.

3. Interpolation (address calculation search)

Das Interpolationsverfahren basiert darauf, daß die Verteilung der Schlüsselwerte bekannt ist.

Annahme: Es gibt einen Algorithmus *f(v,v_1,v_2)* der angibt auf welchem Bruchstück des Weges zwischen v_1 und v_2 der gesuchte Wert *v* liegt.



Dieser Wert muß (wieder per Tabelle) in eine Blockadresse umgewandelt werden:

$$B_i \leftarrow i = \lceil n \cdot f(v, v_1, v_2) \rceil$$

Der Schlüssel des ersten Satzes des so ermittelten Index-Blocks wird dann mit dem gesuchten Wert *v* verglichen dann wird wie bei der binären Suche weiter verfahren:

$v < v_1$: weiter mit den Blöcken $B_1 \cdots B_{i-1}$.

$v \geq v_1$: weiter mit den Blöcken $B_i \cdots B_n$.

Es kann gezeigt werden, daß dieses Verfahren

$\boxed{3 + \log_2 \log_2 n}$ Blockzugriffe

benötigt.

Dies sind bei dem obigen Beispiel

6 Blockzugriffe statt 13

wie bei der binären Suche.

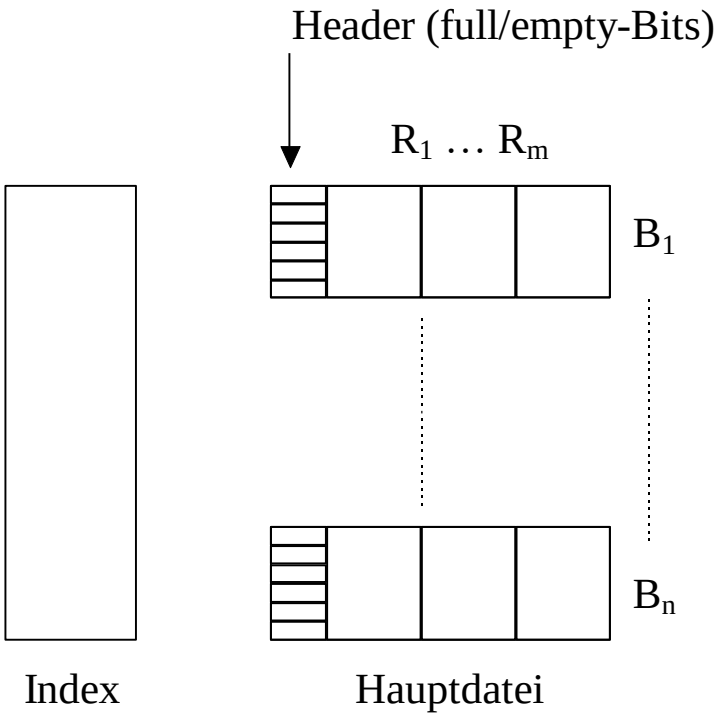
INDEX MIT UNPINNED RECORDS

- Suchen
- Einfügen
- Löschen
- Modifizieren

Annahme:

- Hauptdatei sortiert, Sätze sind unpinned.
- (Sparse) Index sortiert, Sätze sind unpinned.

1. Suchen



Suche v_1:

- Finde im Index nach dem Block dessen erster Satz einen Schlüssel v_2 hat, der v_1 überdeckt.
- Suche in diesem Block nach dem Satz mit dem Schlüssel v_1. (full/empty-Bits beachten!)

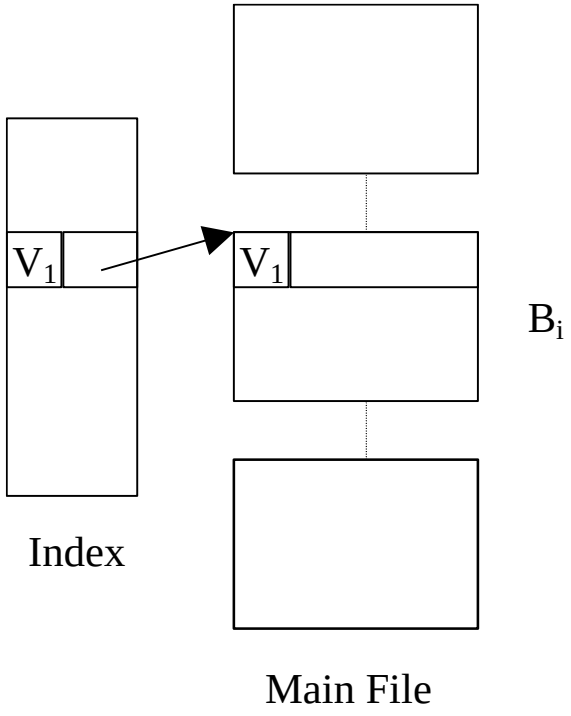
2. Modifizieren

Um den Satz mit dem Schlüssel v_1 zu modifizieren, suche zuerst den entsprechenden Satz.

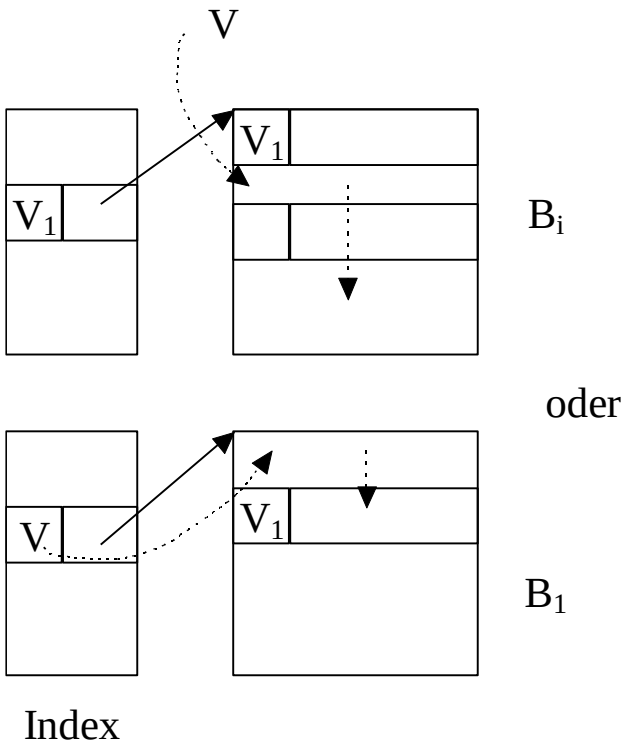
- Falls die Modifikation den Schlüssel betrifft, behandle die Modifikation als **Einfügen** und **Löschen**.
- Falls nicht, modifiziere die Daten und schreibe den Block zurück.

3. Einfügen

Zum Einfügen eines Satzes mit dem Schlüssel v finde den entsprechenden Block B_i. Ist v kleiner als der Schlüssel des ersten Blocks im Index, dann nimm den ersten Block.

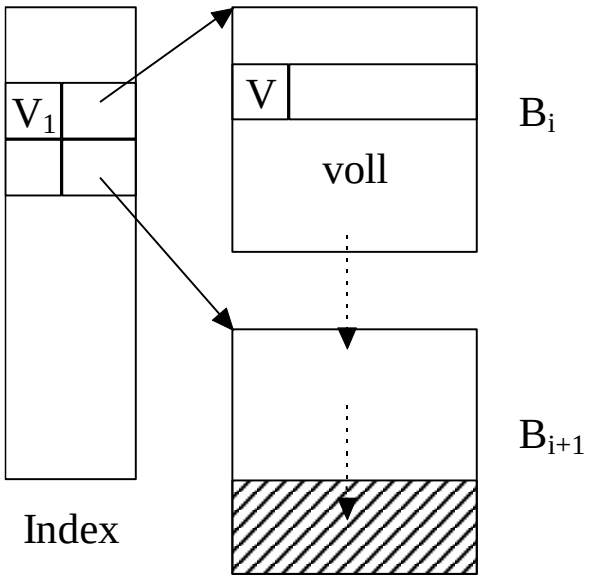


1. Suche die passende Stelle zum Einfügen in B_i. Verschiebe alle Sätze ab dieser Stelle um einen Subblock nach rechts. Füge den Satz ein.



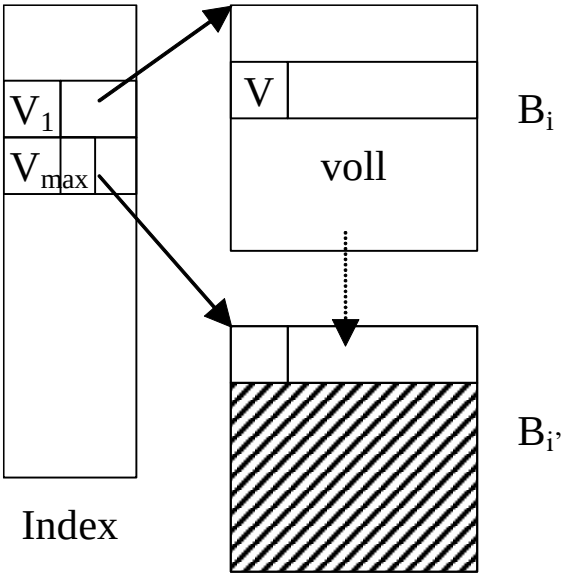
2. Ist in B_i kein Subblock mehr frei, dann bleibt der Satz mit dem größten Schlüssel v_{max} übrig. Ist im nächsten Block B_{i+1} noch ein Subblock frei , dann

   i) füge den Satz v_{max} in B_{i+1} ein,

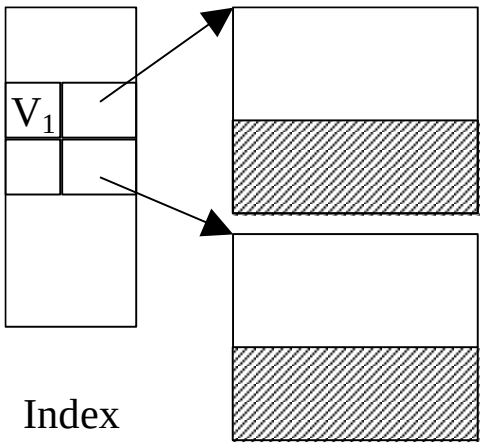


   wenn kein Block mehr frei ist, dann →

ii) nimm einen neuen Block $B_{i'}$, füge den Satz in $B_{i'}$ ein. Füge einen Satz (v_{max},$b_{i'}$) hinter dem Eintrag für B_i in den Index ein.



alternativ:

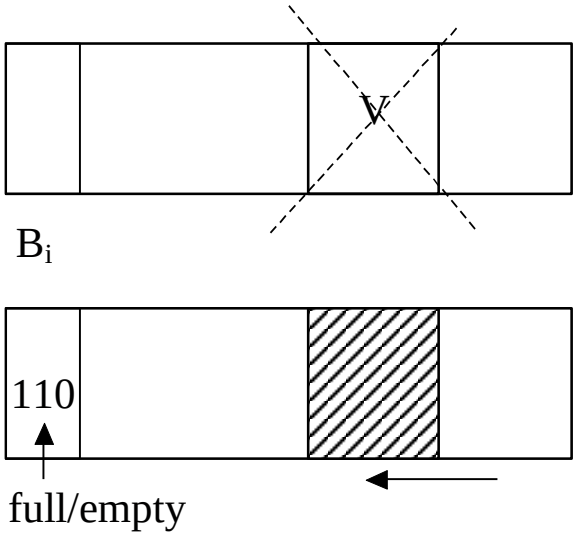


4. Löschen

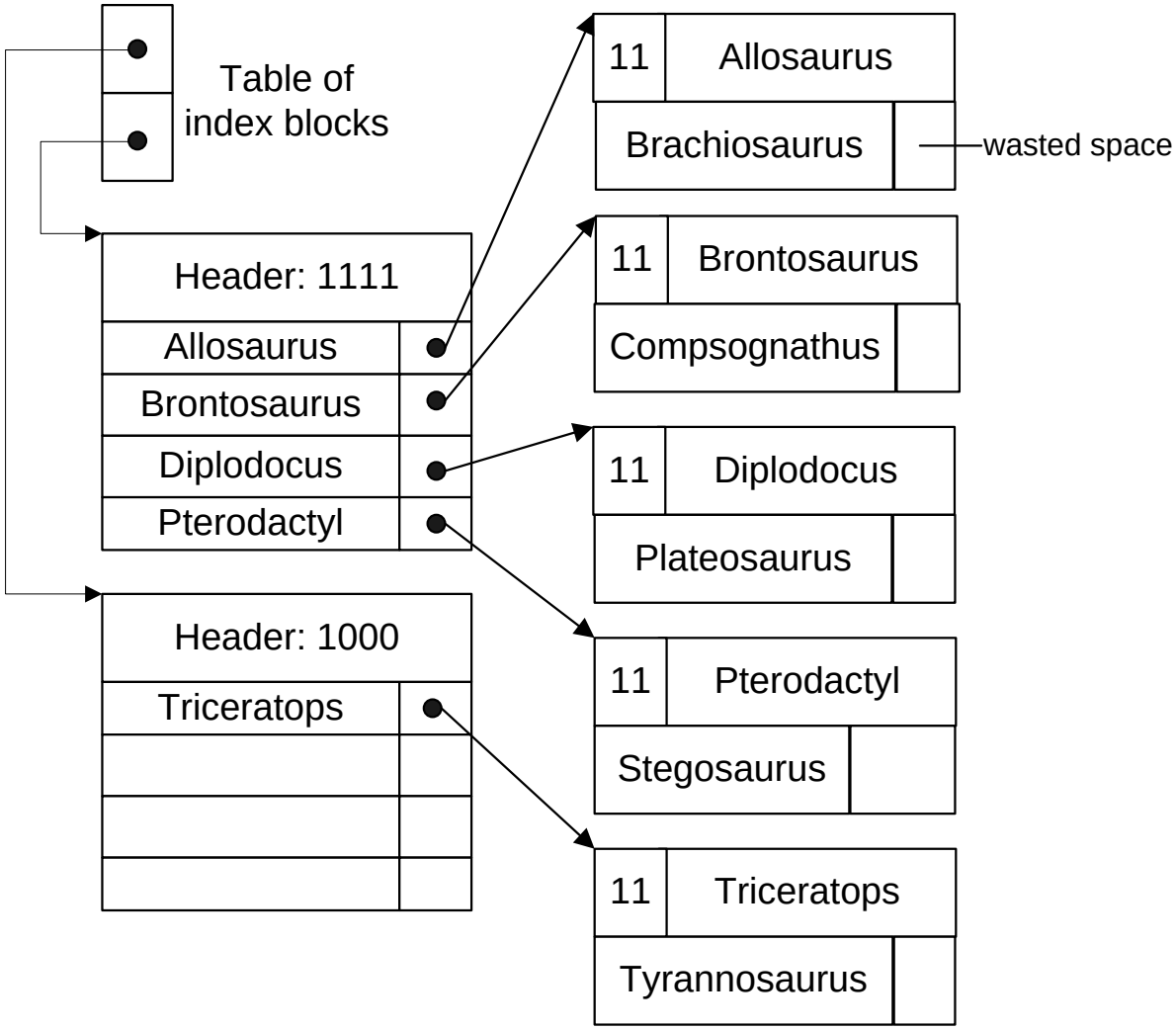
a) Finde den Block, der den gesuchten Satz enthält und lösche den Satz.

b) Ist der Block leer, dann gib ihn frei und lösche den dazugehörigen Indexeintrag.

c) Ist der Block nicht leer, rücke die nachfolgenden Sätze nach links, um die Lücke zu füllen. War der gelöschte Satz der erste Satz, dann ändere den Schlüsselwert im Indexeintrag.



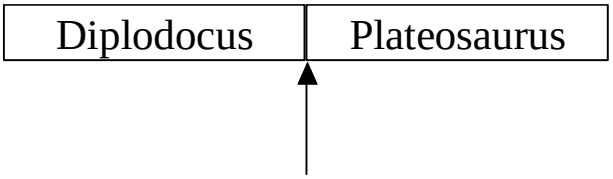
Ausgangszustand der Dinosaurier Datenbank



Beispiel Dino-Daten:

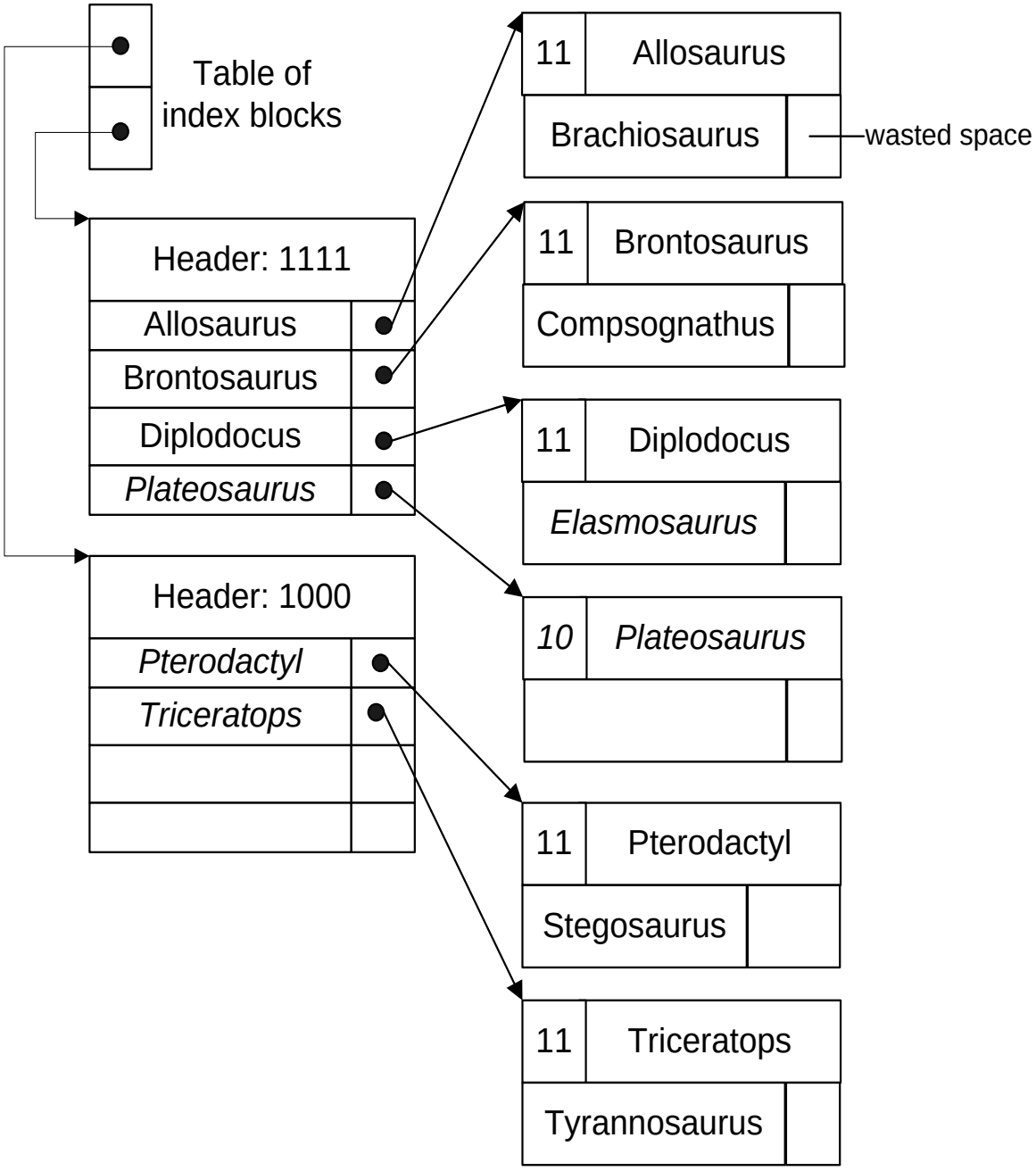
Einfügen von „Elasmosaurus“

1. Suche den Index nach „Elasmosaurus“
   → „Diplodocus“
2. Nimm den entsprechen Block der Hauptdatei (Nr. 3).
3. Durchsuche den Block:



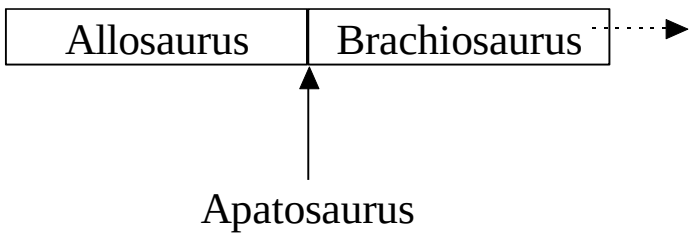
4. Füge „Elasmosaurus“ vor „Plateosaurus“ ein
   → Block ist voll.
5. Finden des nächsten Blocks über den Index, Prüfen des 4ten Blocks → voll.
6. Neuen Block bilden.
7. Indexeintrag erzeugen.

Ergebnis des Einfügens von „Elasmosaurus“



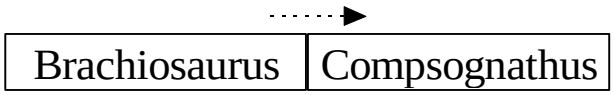
Ändern von „Brontosaurus“ zu „Apatosaurus“

1. Suche im Index nach „Brontosaurus“.
2. Lösche den Satz (weil der Schlüssel geändert wird)
3. Bewege „Compsognathus“ nach links; setze die full/empty-Bits auf „10“.
4. Modifiziere den Indexeintrag für Block 2 zu „Compsognathus“.
5. Einfügen von „Apatosaurus“:
6. Suche im Index nach „Apatosaurus“
   → „Allosaurus“
7. Nimm den Block Nummer 1.
8. Füge „Apatosaurus“ ein; Block ist voll.



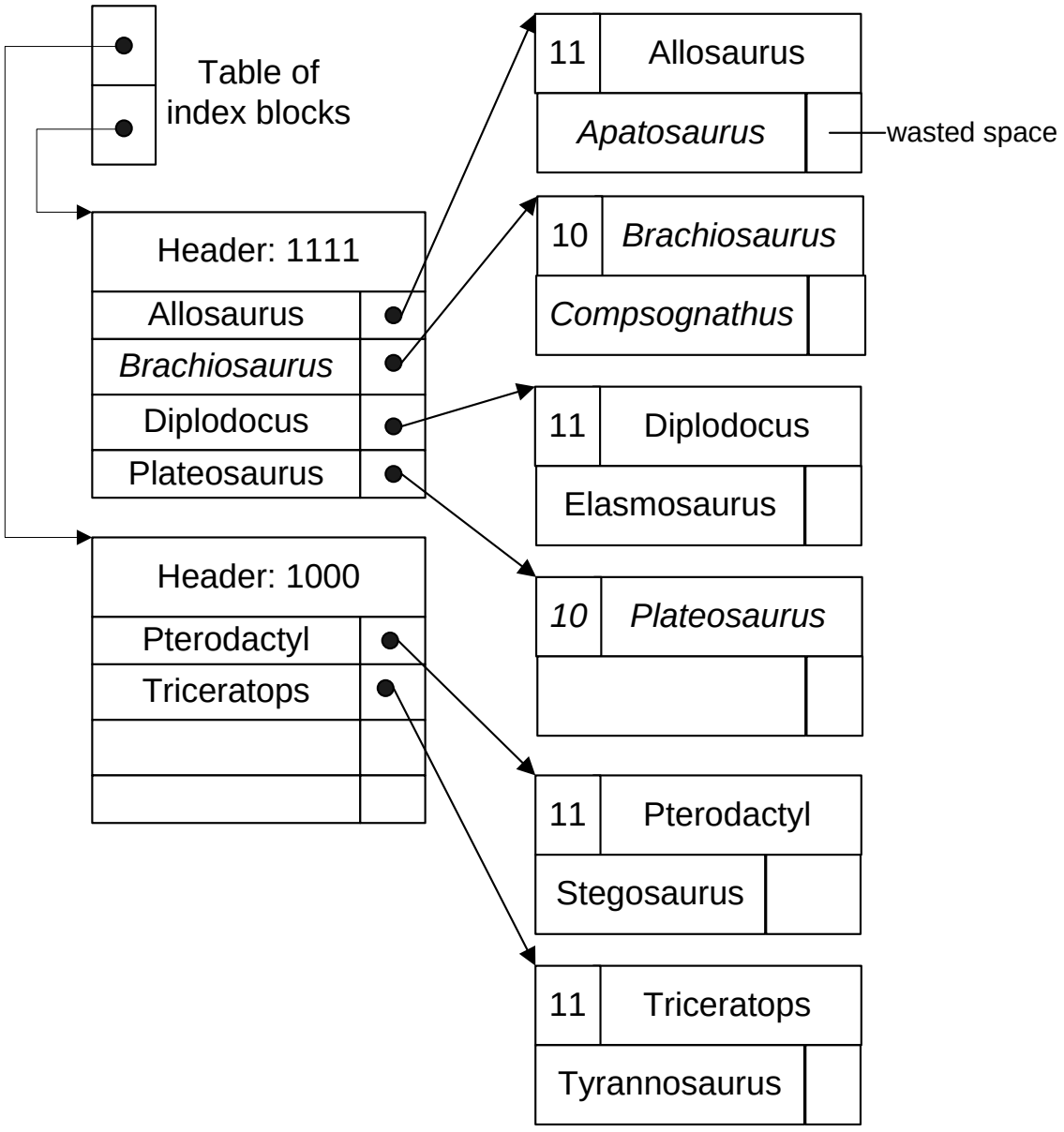
9. Prüfen des nächsten Blocks (über Index); Platz ist vorhanden.

10. Füge den Satz „Brachiosaurus“ ein; setze die full/empty-bits auf „11“



11. Modifiziere den Indexeintrag von Block 2 zu „Brachiosaurus“.

Ergebnis: Ändern von „Brontosaurus“ zu „Apatosaurus“

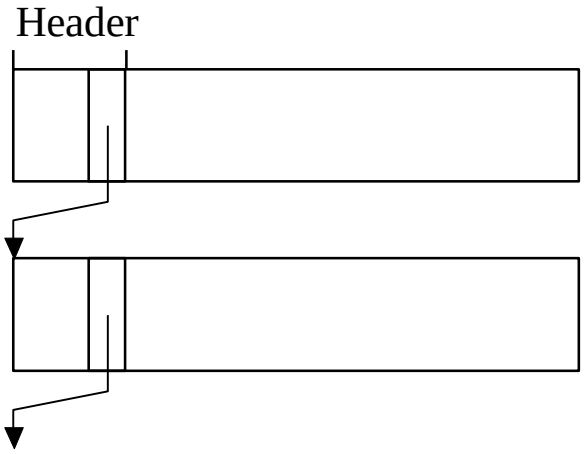


Verkettung von Blöcken

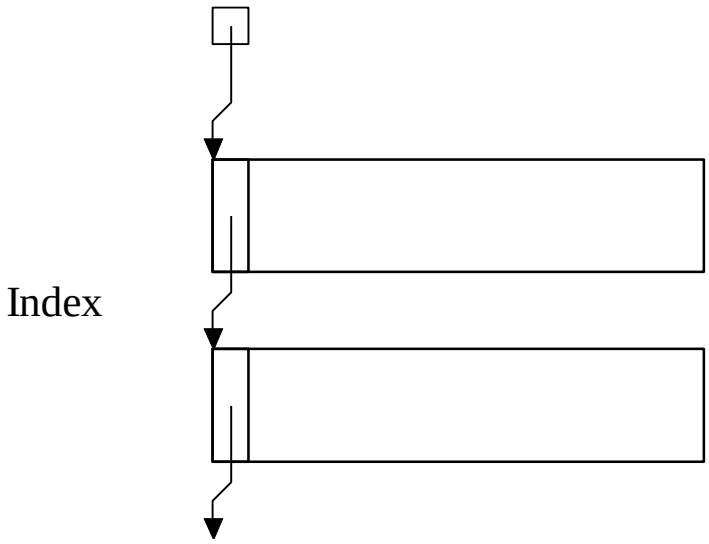
Manchmal muß bei der Operation Einfügen der „nächste“ Block der Hauptdatei aufgesucht werden

→ Index

Alternativ dazu können die Blöcke untereinander mit einem Zeiger im Header verkettet werden:



Wenn dies mit ebenfalls mit dem Index gemacht wird, benötigt man für die lineare Suche im Index **keine Tabelle** mehr, sondern nur noch einen Zeiger auf den ersten Block des Index:



INDEX MIT PINNED RECORDS

Annahme:

- Sätze der Hauptdatei sind pinned.

Folgen:

- Die Sätze innerhalb eines Blocks können nicht sortiert gehalten werden.
- Es ist schwierig, sicher zu stellen, daß die Sätze eines Blocks vor den Sätzen des folgenden Blocks liegen.

Mögliche Lösung:

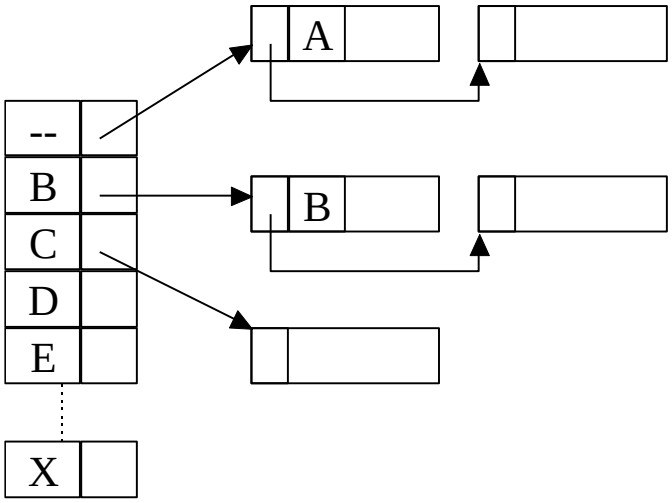
- Anfangen mit der gleichen Organisation wie bei unpinned Records.

- Jeder Block der Hauptdatei wird als erster Block eines **Buckets** gesehen.

- Bei dem Einfügen von Sätzen werden zusätzliche Blocks zum Bucket hinzugefügt; die Blocks werden untereinander durch Zeiger verkettet.

- Es wird ein leerer Block am Anfang erzeugt um ein Bucket für die Sätze zu haben, deren Schlüssel kleiner sind, als der des ersten Satzes der Hauptdatei. Der entsprechende Indexeintrag hat keinen Schlüssel.

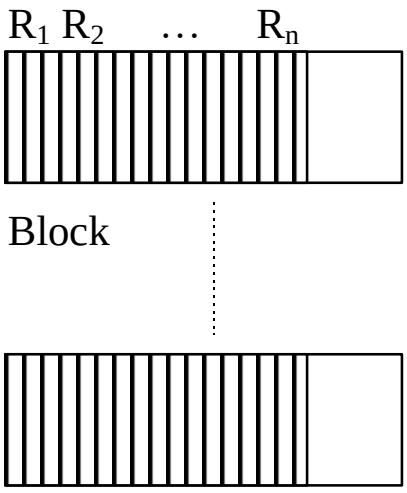
- Der Index ändert sich bei dieser Organisation **nie.**



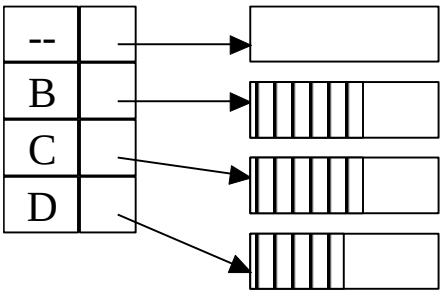
Index,
fix bis zur Datei-Reorganisation

Initialisierung

- Sortieren der Datei.
- Verteilen der Records auf Blöcke; dabei kann immer etwas Platz für Inserts freigelassen werden.

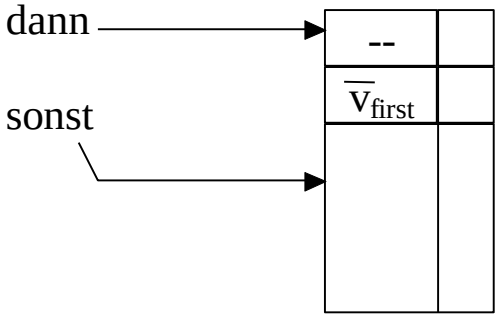


- Erzeugen des Index

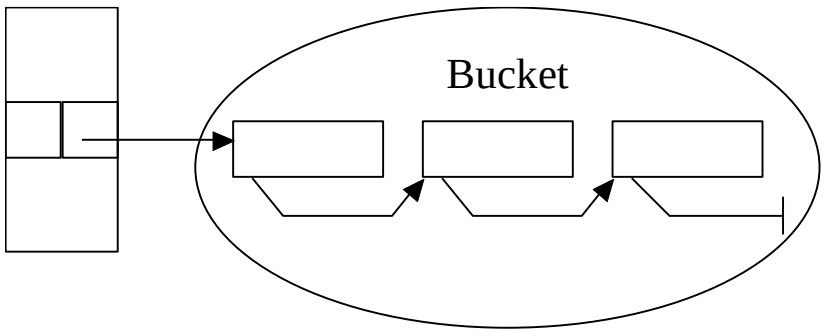


1. Suchen von V

- Suchen des Index Records mit $\bar{v} > v$.
- Falls $v < \bar{v}_{first}$



- Folge dem entsprechenden Zeiger zum Bucket.



- Durchsuche das Bucket.

2. Modifikation:

- Identisch mit dem Verfahren für Modifikationen bei unpinned Records.

3. Einfügen:

1. Finde den entsprechenden Indexeintrag.

2. Durchsuche die Kette der Blöcke (das Bucket) unter diesem Eintrag nach einem freien Subblock.

3. Ist einer gefunden, dann trage den Satz dort ein. Sind alle Blöcke voll, dann nimm einen neuen Block, trage den Satz dort in den ersten Subblock ein und hänge den Block ans Ende der Kette.

4. Löschen:

- Finde den Block, der den Satz enthält und lösche den Satz unter Benutzung von deletion-Bits.

Beispiel Dino-Daten:

Abb.: Sparse Index, pinned records

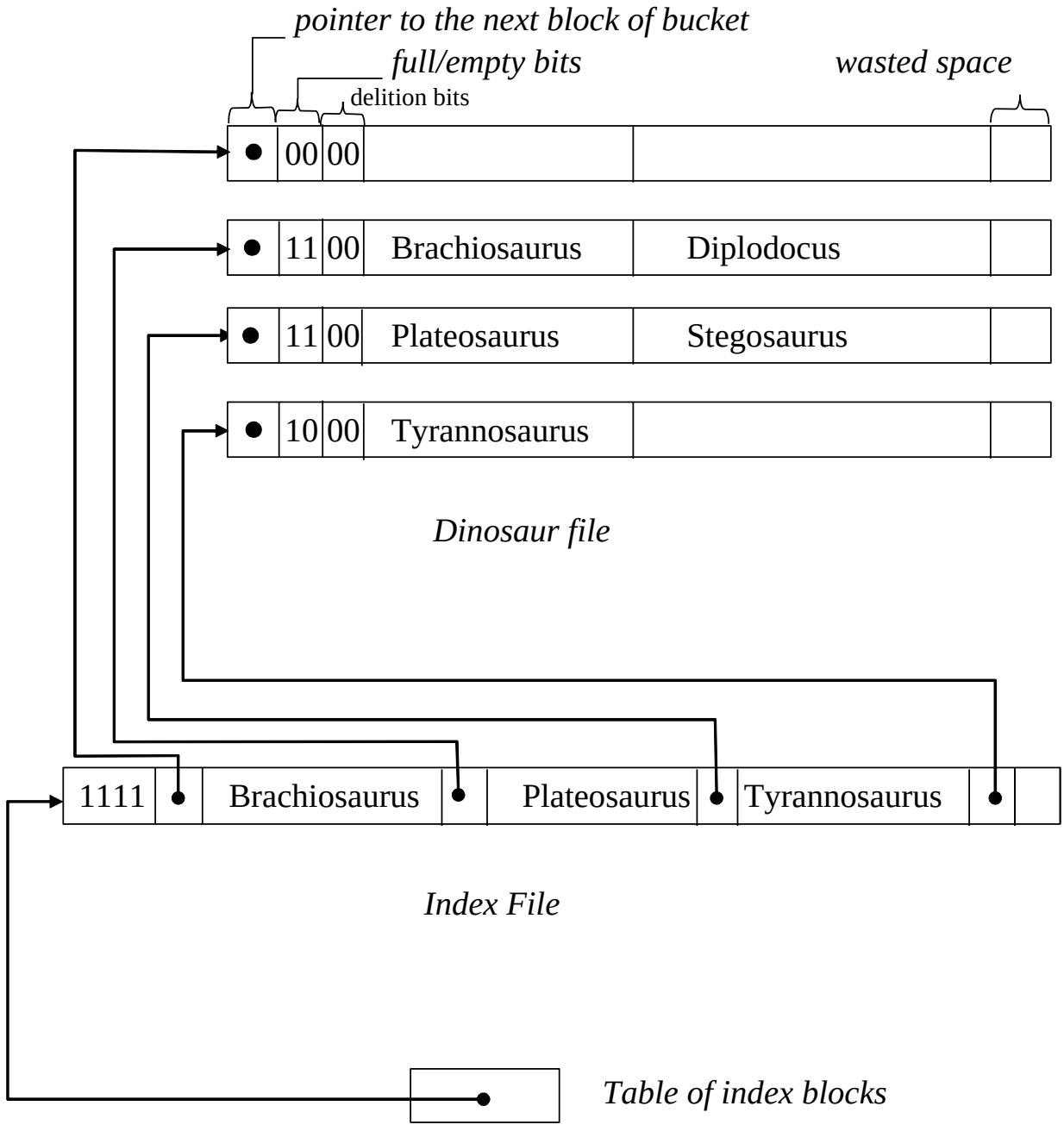


Fig. 2.7. Initial file organisation.

1. Einfügen von

   i) Allosaurus

      - Dieser Satz kommt in den ersten Subblock des anfänglich leeren ersten Buckets.

   ii) Brontosaurus

      - Dieser Satz kommt in das zweite Bucket. Da der einzige Block dieses Buckets voll ist, wird ein neuer Block angelegt

   iii) Compsognathus

      - Dieser Satz kommt ebenfalls in das zweite Bucket, und dort in den zweiten Subblock des zweiten Blocks.

   iv) Elasmosaurus

      - Auch dieser Satz kommt in das zweite Bucket. Da alle Blöcke voll sind, wird ein dritter Block benötigt.

   v) Pterodactylus

      - Dieser Satz gehört in Bucket Nummer 3, und kommt dort in einen neuen Block.

vi) Triceratops

- Dieser Satz kommt in das dritte Bucket.

2. Ändern

i) Brontosaurus → Apatosaurus

- Der Satz für Brontosaurus im zweiten Block wird gelöscht.
- Der Satz für Apatosaurus wird im ersten Bucket eingefügt, dort belegt er den zweiten Subblock der ersten Blocks.

nach Hinzufügen von: Allosaurus, Brontosaurus, Compsognathus, Pterodactyl, Triceratops, Elasmosaurus

Ändern von Brontosaurus zu Apatosaurus

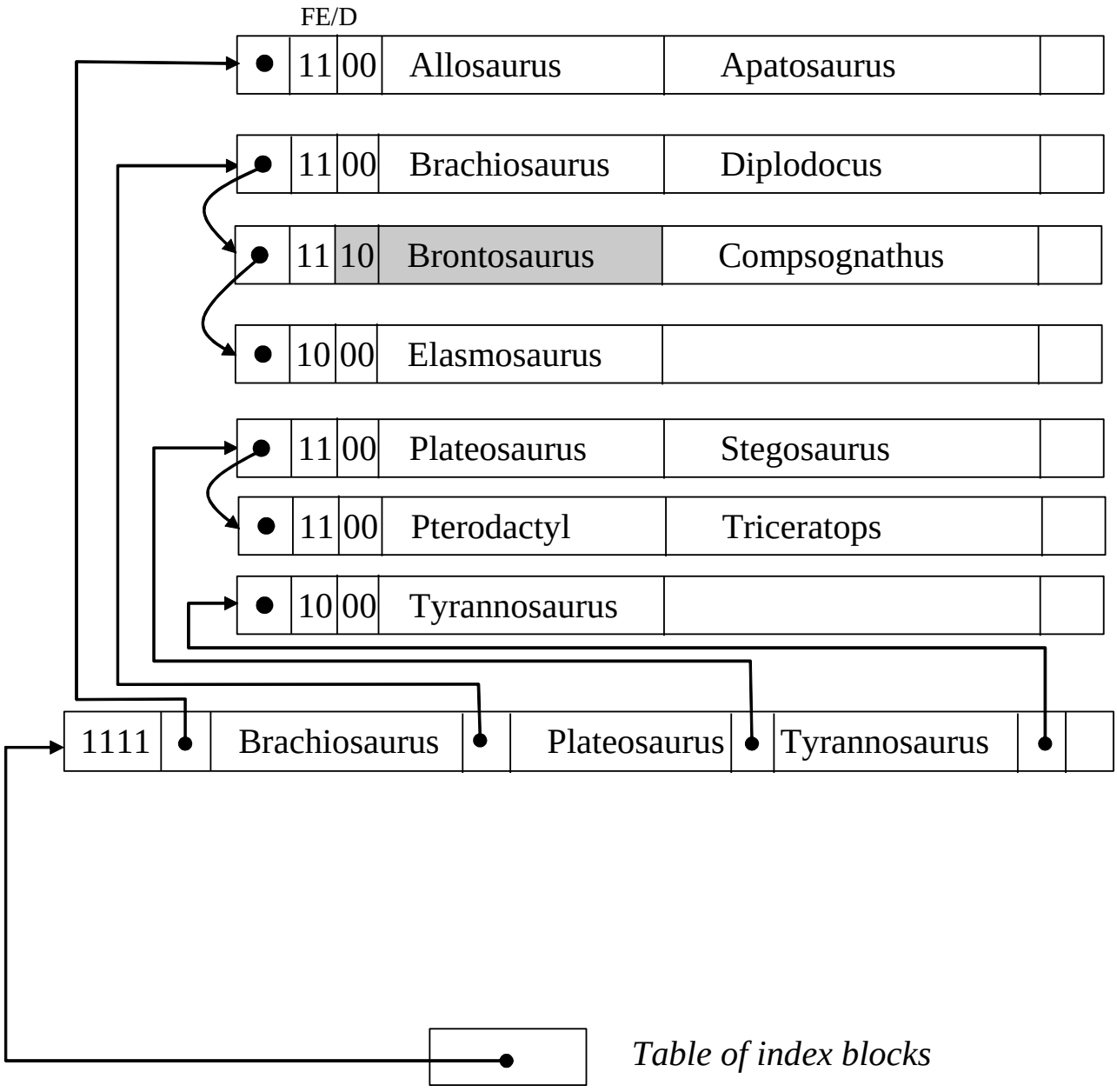


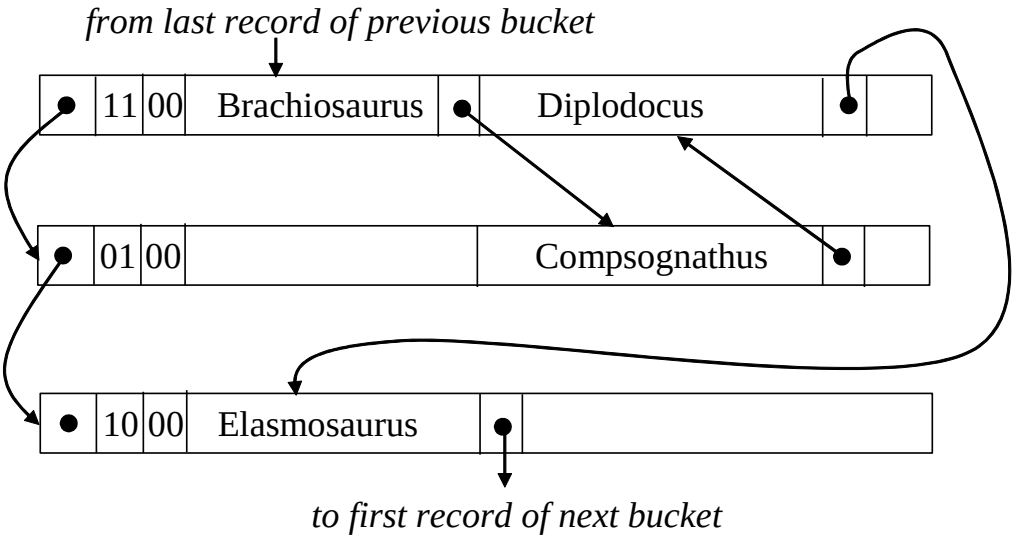
Fig. 2.8. Final file organisation.

Zusätzliche Verknüpfungen

- Es ist nützlich, auch bei pinned Records die Indexblöcke untereinander zu verknüpfen.

- Auch die Buckets können in der richtigen Reihenfolge verknüpft werden, dies kann z.B. mit einem weiteren Zeiger im Header geschehen, oder der Zeiger im letzen Block jedes Buckets zeigt nicht weiter auf „Null“ sondern auf den ersten Block des folgenden Buckets (Ein Bit im Header zeigt dann an, ob der Zeiger das nächste Bucket oder den nächsten Block des selben Buckets referenziert).

- Um die Ordnung der Records untereinander wiederherzustellen kann zu jedem Record ein Zeiger hinzugefügt werden, der auf den nächsten Record in der Sortierreihenfolge verweist.


from last record of previous bucket
11
00
Brachiosaurus
Diplodocus
01
00
Compsognathus
10
00
Elasmosaurus
to first record of next bucket