

6.5 Organisation zeitabhängiger Daten

Überblick

1. Einführung: Motivation und Problemstellung
2. Bisherige Methoden
 - 2.1 Datenduplizierende Verfahren
 - 2.2 Multi-Version B-Tree (MVBT)
3. Anfragebearbeitung ohne Duplikate
 - 3.1 Traditioneller Ansatz
 - 3.2 Referenzmethode
 - 3.3 Link-Methode

Was kann mit Zeit in Beziehung sein?

- Attributwerte
- Datensätze
- Objekte
- Mengen von Datensätzen
- Datenobjekte des Schemas

Welche Zeitachsen gibt es in Datenbanken?

- absolute Zeit
 - Änderungen können sich auf den zukünftigen, gegenwärtigen und historischen Zustand der Datenbank auswirken.
 - hält fest, wann ein Fakt in der realen Welt wirksam wurde.
- Transaktionszeit
 - Änderungen wirken nur auf den gegenwärtigen Zustand der Datenbank ein.
 - hält fest, wann ein Fakt in der Datenbank wirksam wurde.
- Beispiel:
 - Vertrag bei einer Versicherung

6.5.1 Einführung

Anwendungen im Bereich Banken & Versicherungen

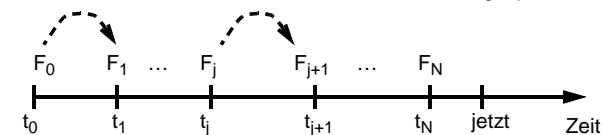
- Kontoführung
 - Anlegen eines neuen Kontos
 - Ändern eines Kontostands (**Erhalten** des alten Wertes)
 - Auflösen eines Kontos
- Anfragen:
 - Kontoauszüge
 - Stand aller Konten am 31.5.95 in der Filiale Leopoldstr.

Multi-Media Anwendungen

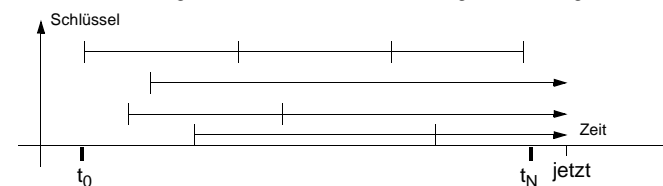
- Speicherung zeitabhängiger Daten
 - Video, Audio, ...

Problemstellung (Transaktionszeit)

- datei-orientierter Blickwinkel:
 - neue Versionsdatei entsteht durch eine Änderungsoperation



- datensatz-orientierter Blickwinkel:
 - Datensatz gehört zu einer zusammenhängenden Menge von Versionen



Datensatz R besteht aus folgenden Komponenten:

- Schlüssel K
- Informationsteil *info*
- Zeitpunkt t_{ins} , an dem der Datensatz in die Datenbank eingefügt wurde oder an dem der Informationsteil *info* geschrieben wurde.
- Zeitpunkt t_{del} , an dem der Datensatz aus der Datenbank gelöscht wurde oder an dem der Informationsteil *info* überschrieben wurde.

Einfügen eines Datensatzes (K,info)

- INC(now);
- Füge den Datensatz $R = (K, \text{info}, t_{now}, *)$ in die Datenbank (F_{now}) ein.

Löschen eines Datensatzes (K,info)

- Falls es ein Datensatz (K, info) im aktuellen Zustand der Datenbank (F_{now}) gibt, d.h. es existiert ein Zeitpunkt t_{ins} , $ins < now$, so daß es $R = (K, \text{info}, t_{ins}, *)$ gibt.
 - INC(now); $R.t_{del} = now$;
 - Zurückschreiben des modifizierten Datensatzes R.

263.

Wunscheigenschaften einer effizienten Lösung

- b = Kapazität einer Seite **!! keine Konstante !!**
- N = Anzahl der Versionen

Zugriffszeit

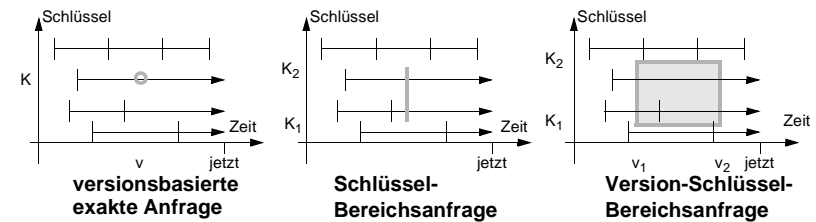
- wie beim B+-Baum (asymptotisch):
 - versionsbasierte exakte Suche: $O(\log_b n)$
 - Schlüssel-Bereichssuche: $O(\log_b n + r)$
 - Änderungsoperationen: $O(\log_b n)$
- Version--Schlüssel Bereichsanfragen ?

Speicheraufwand

- $O(N)$ = linear in der Anzahl der Versionen

Anforderungen

- effiziente Unterstützung von Anfragen



- effiziente Unterstützung von Änderungsoperationen
 - Einfügen, Abändern vorhandener Datensätze, Löschen
- niedriger Speicheraufwand für die Versionsdateien
- Annahme: Anfragen oft auf neuen Versionen

6.5.2 Bisherige Ansätze

1. Naiver Ansatz:

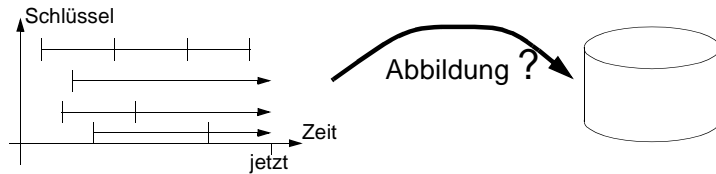
- Änderungsoperation ==> Kopieren der kompletten Datei
 - + effiziente Abfragebearbeitung
 - hohe Speicherkosten
- Spezialfall insertion-only: $O(N^2)$

2. Naiver Ansatz:

- Datensatz: $(K, in_time, del_time, info)$
- verwende mehrdimensionale Indexstruktur (z. B. R-Baum)
 - + geringe Speicherkosten
 - hohe Abfragekosten: $> O(\log N)$ bei exakter Versionsanfrage

6.5.2.1 Datenduplizierende Methoden

- geeignet für Klasse von Zugriffsstrukturen (B⁺-Bäume)



- zweidimensionaler Datensatz mit Ausdehnung: $\langle K, \text{in_time}, \text{del_time}, \text{info} \rangle$

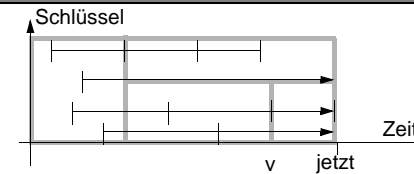
Idee:

- unterteile den Datenraum vollständig in nicht-überlappende Regionen
- ordne jeder Region eine Seite zu und speichere die zweidimensionalen Datensätze in den Seiten, so daß
Satz schneidet Seitenregion \implies abspeichern in der entsprechenden Seite

Was passiert beim Überlauf einer Seite?

Zeit-Spalten

Zeit-Spalten



Wähle einen Zeitpunkt t_{split} und teile die Datensätze aus der Seite L auf die Seiten L und R folgendermaßen auf:

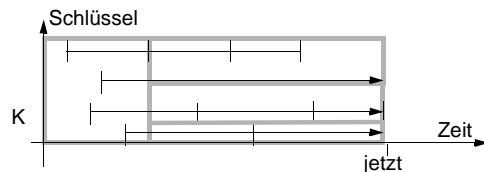
- alle Datensätze R mit $R.t_{\text{del}} < t_{\text{split}}$ verbleiben in der Seite L.
- alle Datensätze R mit $R.t_{\text{ins}} \geq t_{\text{split}}$ kommen in die neue Seite R.
- alle anderen Datensätze verbleiben in L und kommen zusätzlich nach R.

Eigenschaften:

- hohe Speicherkosten durch Erzeugung der Duplikate
- + niedrige Anfragezeiten (durch Clustering)

Schlüssel-Spalten

Schlüssel-Spalten



Wähle einen Schlüssel K_{split} und teile die Datensätze aus der Seite L auf die Seiten L und R (analog zum B+-Baum):

- alle Datensätze R mit $R.K < K_{\text{split}}$ verbleiben in der Seite L.
- alle Datensätze R mit $R.K \geq K_{\text{split}}$ kommen in die neue Seite R.

- + niedrige Speicherkosten
- hohe Anfragekosten

Time-Split-B-Tree (TSBT)

- Lomet & Salzberg 89 / 90 / 93

- Regeln für das Aufspalten beim TSBT:

Falls historische Datensätze existieren:

1. Zeit-Spalten: Zeitpunkt der letzten Änderungsoperation in der Seite L.
2. #Datensätze $> 2/3 \cdot b \implies$ Schlüssel-Spalten

- Nachteile des Verfahrens

- logisches Löschen durch leere Stellvertreter-Datensätze
- keine garantierte Clustering alter Versionen
- gesonderte Behandlung von Index- und Datenseiten
 - komplizierte Algorithmen
 - worst-case Verhalten ?

6.5.2.2 Multiversion B-Tree (MVBT)

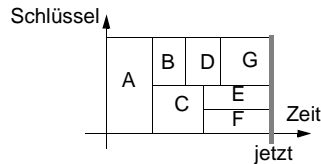
- daten-duplizierenden Methode

Entwurfsziele:

- Anfrageleistung: (asymptotisch) wie beim B+-Baum
- Speicherkosten: $O(N)$ = linear in der Anzahl der Versionen

notwendige Bedingungen:

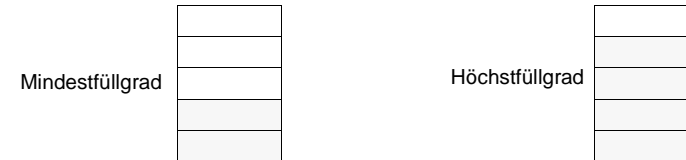
- eine Seite P mit Region $I_T \times I_K$
für alle $v \in I_T \implies$ es gibt d Datensätze zur Version v in P



- kein Aufspalten frisch aufgespaltener Seiten (**starke Versionsbedingung**)

Behandlung eines Überlaufs

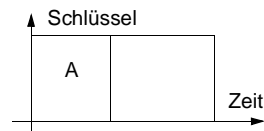
- $d = b/5$ (b = Kapazität der Seiten)
- Zeit-Spalten: verwende aktueller Zeitpunkt (*now*)
 - #Datensätze $> 4d \implies$ Schlüssel-Spalten
oder
 - #Datensätze $< 2d \implies$ beseitige diese "unterfüllte" Seite.
- "unterfüllte" Seite: Verschmelzen mit einer Partnerseite
 - #Datensätze $> 4d \implies$ Schlüssel-Spalten.



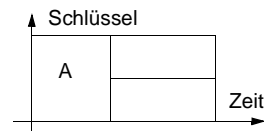
- starke Versionsbedingung:
eine neue Seite besitzt zwischen $2d$ und $4d$ Datensätze.

Möglichkeiten der Reorganisation

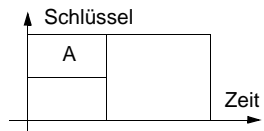
reines Zeit-Spalten



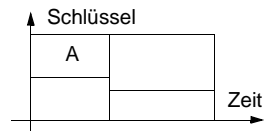
Zeit- & Schlüssel-Spalten



reines Verschmelzen



Verschmelzen & Schlüssel-Spalten



Worst-Case Leistung des MVBT

- m_i = #Datensätze in der i-ten Version
- #Diskzugriffe für eine exakte Versionsanfrage in Version i:

$$2 + \lceil \log_d m_i \rceil$$

- #Diskzugriffe für die i-te Änderungsoperation:

$$2 + 5 \lceil \log_d m_i \rceil$$

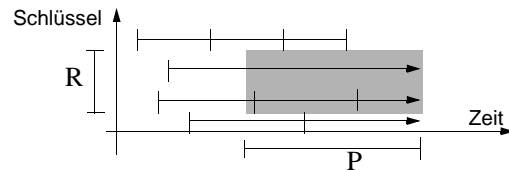
- Speicherplatzkosten: $O(N)$

- Für B-Bäume ist dieser Ansatz asymptotisch optimal:
Ein B-Baum zur Verwaltung lediglich einer Version zeigt kein besseres Leistungsverhalten!

6.5.3 Anfragebearbeitung ohne Duplikate

Version-Schlüssel-Bereichsanfragen:

- Finde alle Datensätze mit Schlüssel K im Bereich R zur Zeitperiode P



erfordert Zugriff auf eine

- Menge von Datensätzen
- Menge von Datenseiten

Depth-First Algorithmus

Algorithmus DF(Entry f; key_range R; time_period P);

N = GetPage(f);

IF (N is a data page) THEN

...

ELSE

FOR EACH entry g in N DO

compute the key_range [low, up) of entry g;

compute the life-span $[t_{ins}, t_{del})$ of g;

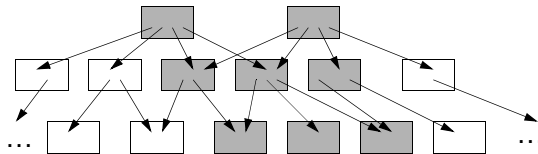
IF $([low, up) \cap R \neq \emptyset)$ AND $([t_{ins}, t_{del}) \cap P \neq \emptyset)$ THEN

DF(g, R, P);

END;

END;

Probleme



- Duplikate in der Antwortmenge
 - sind oft unerwünscht
- Duplikate im Index
 - führen zu erheblich höheren E/A-Kosten

Wie kann der Zugriff auf Duplikate vermieden werden?

6.5.3.1 Traditionelle Ansätze

- Sortieren
- Hashtabelle für Antworten und qualifizierende Indexeinträge

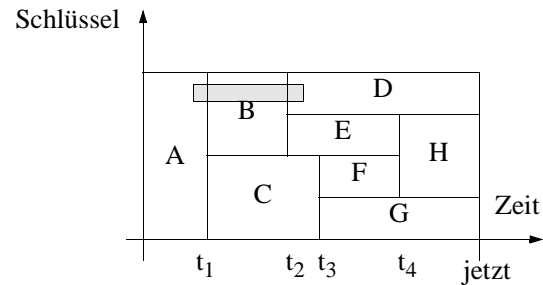
Zusatzaufwand

- Speicherplatz
- CPU-Zeit

6.5.3.2 Referenzpunktmethode

Idee:

- Teste für jeden Eintrag, ob der zugehörige Referenzpunkt in der Seitenregion liegt.

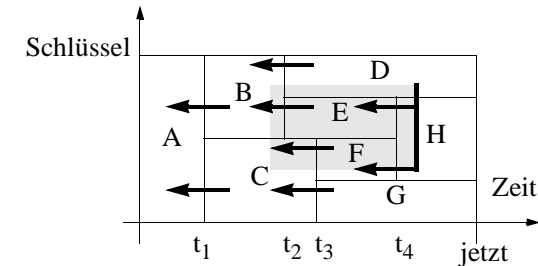


- Vorteil: anwendbar für viele Zugriffsstrukturen
- Nachteil: hoher Aufwand für das Durchsuchen des Index

6.5.3.3 Link-Methode

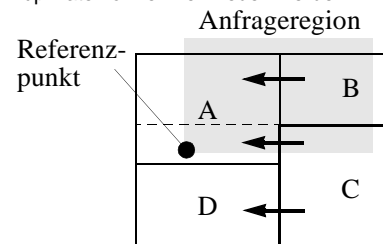
Idee:

- zuerst: Schlüssel-Bereichsanfrage mit dem rechten Rand des Suchbereichs
- danach: verfolge Zeiger zu historischen Vorgängerseiten



- wichtige Eigenschaft für B-Baum basierte Verfahren:
 - es gibt höchstens 2 historische Vorgängerseiten

- Zugriff auf Duplikate können vermieden werden



- Aufwand für eine Zeit-Schlüssel-Bereichsanfrage
 1. Aufwand für die Schlüssel-Bereichsanfrage:
 $O(\log_b m_v + a_1 / b)$
 2. Aufwand für das Verfolgen der Links
 $O(vp / b)$
- Link-Methode nicht für den R-Baum geeignet
 - Überlappung der Seitenregionen
 - keine vollständige Partitionierung