Proseminar Datenbanksysteme

Universität Innsbruck — Institut für Informatik

Bottesch R., Hupfauf B., Kelter C., Mayerl M., Moosleitner M., Peintner A., Zangerl D.

13.12.2022

Übungsblatt 9 – Lösungsvorschlag

Diskussionsteil (im PS zu lösen; keine Abgabe nötig)

a) ★ Begründen Sie warum die Chained I/O Zugriffsmethode im Durchschnitt beim Lesen von mehreren Blöcken schneller ist als die Random I/O Zugriffsmethode.

Lösung ✔

Weil die Random I/O Zugriffsmethode den Arm bei jedem Speicherzugriff neu positionieren muss.

b) ★ (Blockweise Speicherung) Wenn Records variabler Länge verwendet werden, kann bei der Berechnung der Speicheradressen nicht mehr mit konstanten Offsets gearbeitet werden. Was muss stattdessen gemacht werden?

Lösung ✔

Eine Möglichkeit wäre das Einführen von Separatoren, welche die Grenzen zwischen Feldern und Records signalisieren.

c) ★ (Auswirkung der Datenanordnung auf Operationen) Füllen Sie Tabelle 1 aus indem Sie ein **+** einsetzen, wenn die Operation effizient möglich ist und ein **-** wenn sie teuer ist (ohne Verwendung von Indizes und Suche nach Primärindex).

	Sortierte Dateien	Unsortierte Dateien
Einfügen		
Suchen		
Löschen		
sortiertes Lesen		

Tabelle 1: Übersicht Datei-Operationen

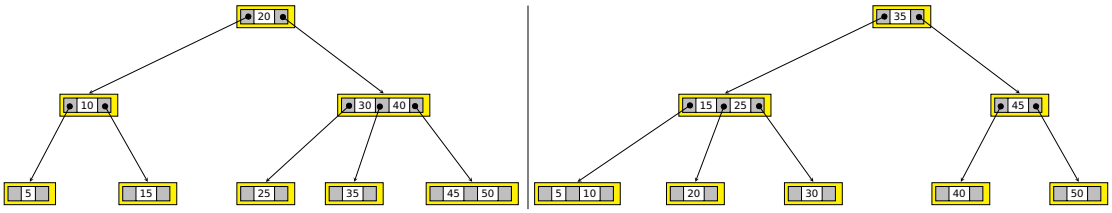
Lösung ✔

	Sortierte Dateien	Unsortierte Dateien
Einfügen	-	+
Suchen	+	-
Löschen	-	-
sortiertes Lesen	+	-

d) ★ Überlegen Sie ob das Einfügen der Werte $X_1, X_2, \ldots X_n$ und Einfügen derselben Werte in umgekehrter Reihenfolge zum selben B-Baum führen.

Lösung ✔

Nein, dies gilt für B-Bäume nicht im Allgemeinen. Betrachten Sie zum Beispiel die verschiedenen B-Bäume die entstehen, wenn Sie die Werte $5, 10, 15, 20, 25, 30, 35, 40, 45, 50$ in aufsteigender bzw. in absteigender Reihenfolge einfügen ($p = 3$).



e) ★ Nehmen Sie an, in der Tabelle `film` (pagila Datenbank) gibt es jeweils einen B-Baum und einen B$^+$-Baum-Index auf das Attribut `film_id`[1]. Erstellen Sie eine Abfrage, die vom B-Baum-Index effizienter abgearbeitet werden könnte (im average case) und eine Abfrage, die effizienter vom B$^+$-Baum-Index abgearbeitet werden kann. Worin unterscheiden sich diese Queries und wieso haben Sie diese gewählt?

Lösung ✔

Range-Queries werden durch B$^+$-Bäume effizienter beantwortet, da alle Daten in den Blättern liegen und diese zumindest in einer Richtung miteinander verbunden sind. Das hat den Vorteil, dass man bei Bereichsabfragen nur durch die Blätter iterieren muss um die gewünschten Daten zu holen und keine Traversierung des Baumes notwendig ist. Eine Beispiel-Abfrage wäre:

```
1    SELECT *
2    FROM film
3    WHERE film_id BETWEEN 3 AND 1000;
```

Punkt-Queries hingegen könnten effizienter über einen B-Baum-Index beantwortbar sein, da die Daten nicht nur in den Blättern, sondern auch in den Knoten liegen und deswegen der B-Baum nicht immer bis zu den Blättern durchsucht werden muss. Eine Beispiel-Abfrage wäre:

```
1    SELECT *
2    FROM film
```

[1] In der Realität werden in modernen DBMS nur B$^+$-Bäume eingesetzt.

```
3    WHERE film_id = 3;
```

Da aber ein B$^+$-Baum viel mehr Indizes aufnehmen kann und somit auch flacher als ein B-Baum ist, macht es nicht wirklich einen Unterschied.

f) ☆ Woran können Sie die Effizienz der Abfrageverarbeitung messen - beispielsweise um zwei Abfragen zu vergleichen oder den Mehrwert einer weiteren Indexstruktur zu evaluieren?

Lösung ✔

Effizienz in Bezug auf Datenbank-Operationen wird generell meist über die Anzahl der Hintergrundspeicherzugriffe (wie oft müssen Datensätze (Blöcke) von der Platte nachgeladen werden). Indexstrukturen können über die $\mathcal{O}$-Notation verglichen werden. Dabei wird die Anzahl der Vergleiche als Vergleichsbasis herangezogen ($\mathcal{O}(logn)$ ist worst case für Suche im B$^+$-Baum). Eine Zeitmessung macht in diesem Fall nicht Sinn, da die Effizienz immer auch von den Daten und der Datenanordnung abhängig ist, sowie auch andere Aspekte eine Rolle spielen, wie z.B. ob eine Abfrage parallelisiert werden kann oder nicht.

g) ☆ Diskutieren Sie, warum moderne Datenbanksysteme einen B$^+$-Baum als Default-Indexstruktur anbieten und nicht z.B. Hash-Indizes zum Einsatz kommen.

Lösung ✔

B$^+$-Bäume sind für eine größere Anzahl an Anwendungsfällen geeignet (z.B. auch Range-Query, welche von Hash-Indizes nicht effizient unterstützt werden können). Auch haben B$^+$-Bäume den Vorteil, dass sie balanciert sind und somit die Suchperformance stabil ist (alle Blätter sind auf gleicher Höhe). Bei Hash-Indizes ist dies nicht der Fall, da deren Performance massiv von der Überlaufstrategie und der Verteilung der Daten (bzw. der Hash-Werte) abhängt und ggf. auch neu gehasht werden muss, falls die Indexstruktur mit der verwendeten Hash-Funktion zu stark gefüllt sind (und damit zu viele Kollissionen auftreten).

h) ☆ Wie viele Vergleiche werden für die Suche eines Eintrags (bzw. des Pointers zum Datensatz) in einem B-Baum der Ordnung p mit n Einträgen benötigt? Bitte vervollständigen Sie dazu Tabelle 2.

	B-Baum	B$^+$-Baum
best case		
worst case		

Tabelle 2: B-Baum vs. B$^+$-Baum

Lösung ✔

	B-Baum	B$^+$-Baum
best case	1	$Baumhoehe * 1$
worst case	$Baumhoehe * (p-1)$	$Baumhoehe * (p-1)$

i) ★★ Daten sollen in einem B-Baum organisiert werden. Die Größe einer Speicherseite der Festplatte betrage 2048 Bytes, die Größe eines Indexeintrages im B-Baum sei 20 Bytes (inkl. 8 Bytes für linken Teilbaumpointer). Berechnen Sie die optimale Ordnung für diesen B-Baum.

Lösung ✔

$2048 - 8 = 2040$ (Platz für Pointer "ganz rechts")
$2040/20 = 102$ (=Anzahl der Paare aus Daten- und Baum-Pointer $< K, P_r >$)
$p = 103$ (jeder B-Baum hat p-1 Datenpointer, daher ist die Ordnung p = 103)

j) ★ In SQL ist es auch möglich, Spalten als "unique" zu kennzeichnen, um zu vermeiden, dass in einer Spalte ein Wert mehrfach auftritt (oder auch in einer Kombination von Spalten). Dies ermöglicht eine zusätzliche Überprüfung zur Datenkonsistenz.

Unique wird in SQL als Index realisiert und kann z.B. beim Erzeugen einer Tabelle mit `UNIQUE (film_id)` erstellt werden. Welche Indexstruktur bietet sich hier an? Bedenken Sie, dass diese Unique-Bedinung für jedes Insert- und Update-Statement überprüft wird.

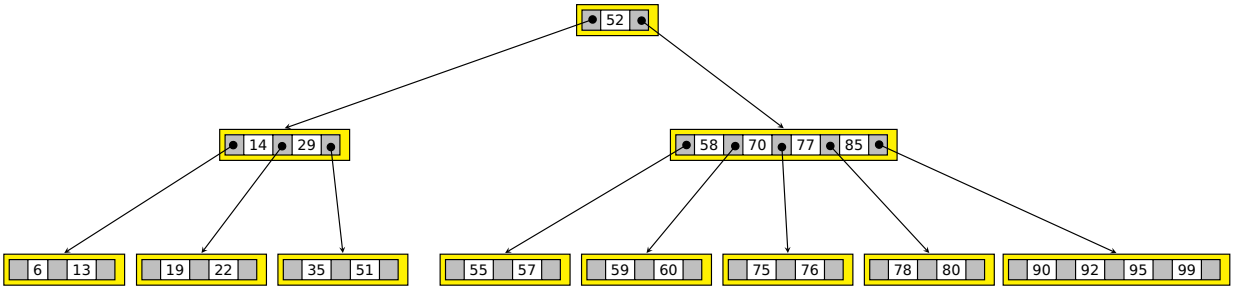
Lösung ✔

Hier bietet sich ein Hash-Index an, da so mit $\mathcal{O}(1)$ überprüft werden kann, ob ein Wert bzw. eine Wertekombination bereits vorhanden ist.

Hausaufgabenteil (Zuhause zu lösen; Abgabe nötig)

Aufgabe 1 (B-Baum Operationen) [4 Punkte]

Gegeben sei der folgende B-Baum der Ordnung $p = 5$.



Führen Sie folgende Operationen jeweils **auf diesem Baum** aus. Zeichnen Sie dabei mindestens

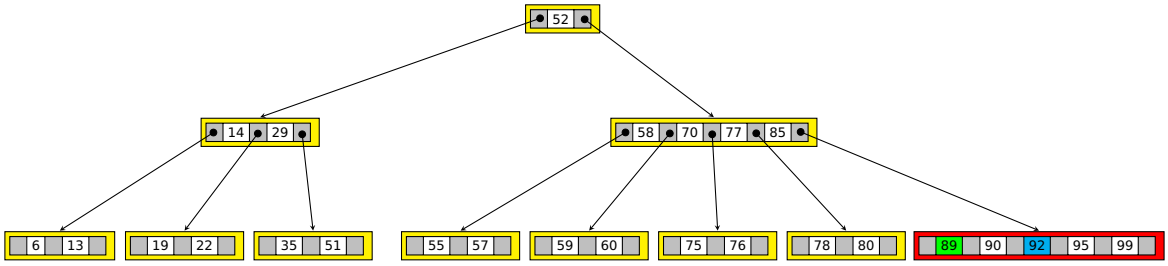
2 Zwischenschritte und das Endergebnis auf und begründen Sie Ihr Vorgehen mit kurzen Kommentaren.

a) *1 Punkt* Einfügen von 89

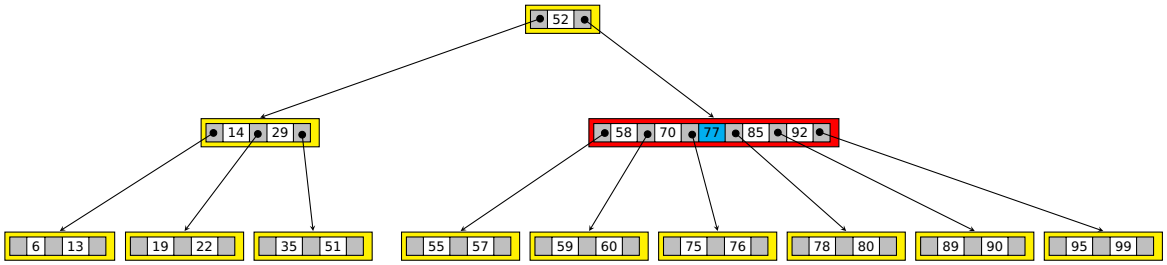
Abgabe

1a.pdf

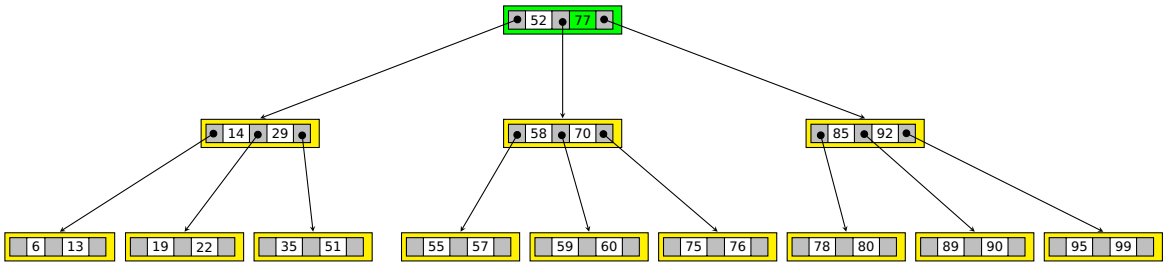
Lösung



Beim Einfügen von 89 kommt es zu einem Überlauf, deshalb kommt die 92 in den Vaterknoten.



Dort kommt es wieder zu einem Überlauf, deshalb muss auch hier das Element in der Mitte in den Vaterknoten (hier die 77).

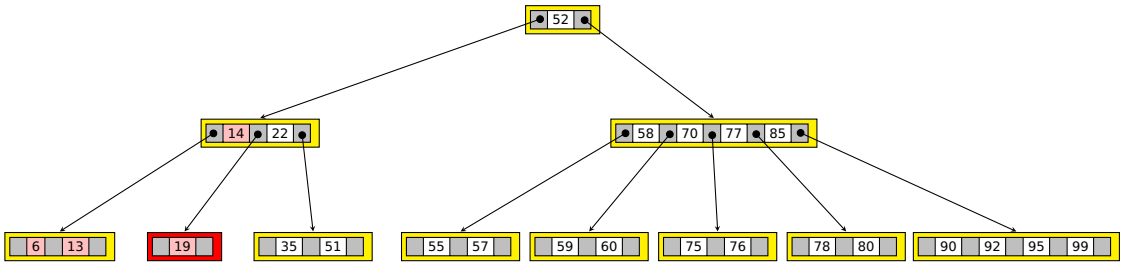


b) *1.5 Punkte* Löschen von 29

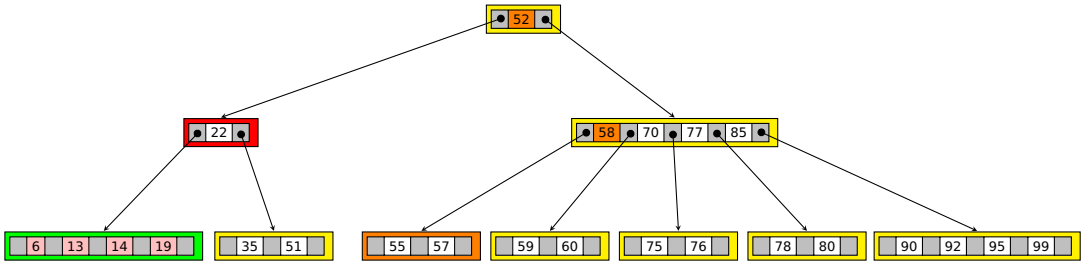
Abgabe

1b.pdf

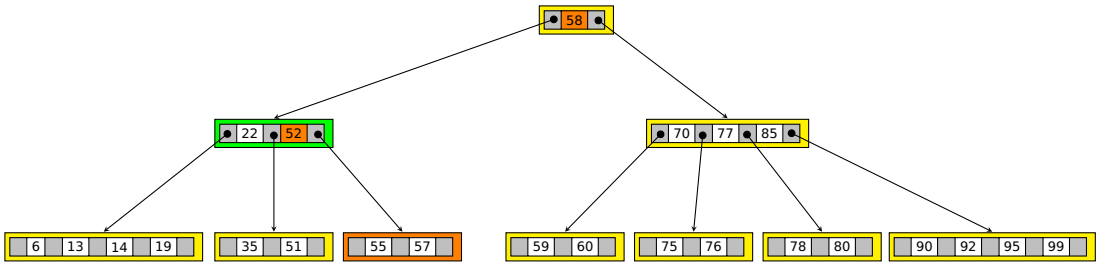
Lösung ✔



Durch das Löschen von 29 kommt es zu einem Unterlauf beim Knoten mit der 19. Dieser Unterlauf wird beglichen, indem es mit der 14 im Vaterknoten und dem linken Nachbarknoten zusammengefasst wird.



Beim Zusammenfassen kommt es zu einem Unterlauf beim Knoten mit der 22. Durch eine große Linksrotation über den Wurzelknoten wird dieser Unterlauf beglichen.

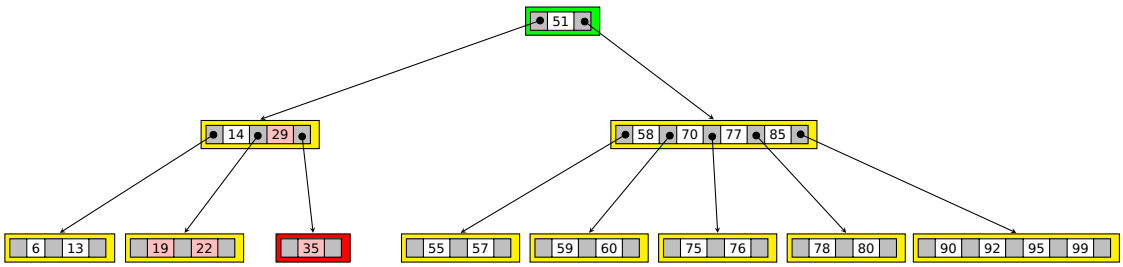


c) *1.5 Punkte* Löschen von 52

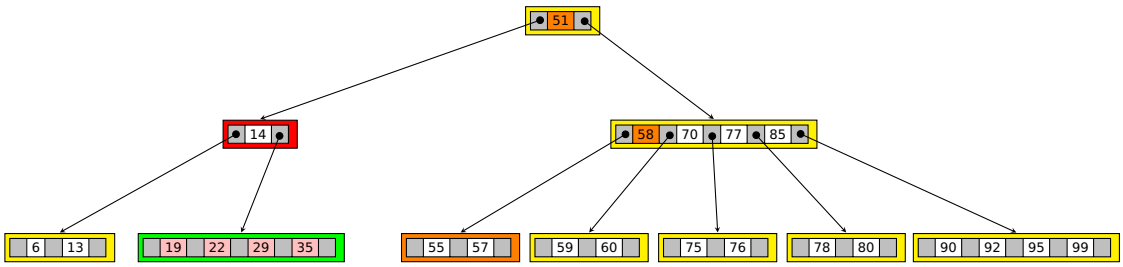
Abgabe ↑

1c.pdf

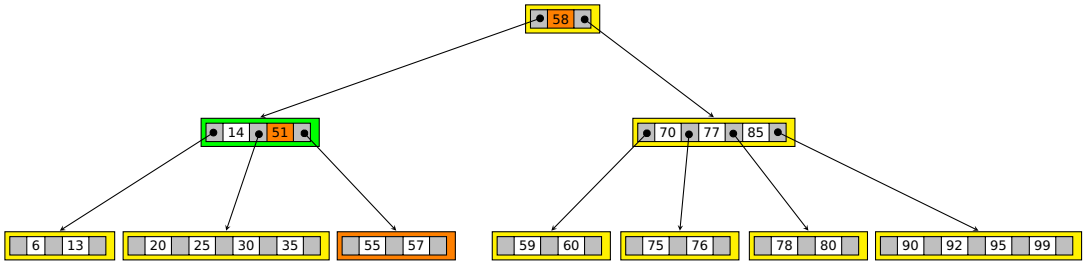
Lösung ✔



Das größte Element im linken Teilbaum wird zum neuen Wurzelknoten. Beim Knoten mit der 35 kommt es dabei zu einem Unterlauf.



Aufgrund des Unterlaufs werden die Blätter mit den Werten $19, 22$ und 35 zusammengefasst.



Beim Zusammenfassen kommt es wiederum zu einem Unterlauf (Knoten mit der 14). Durch eine große Linksrotation über den Wurzelknoten wird dieser Unterlauf beglichen.

Aufgabe 2 (Indexstrukturen in der Praxis) [6 Punkte]

Das Ziel dieser Aufgabe ist zu analysieren, wie Indizes in der Praxis funktionieren, welche Verbesserungen sie mit sich bringen und wie man diese am besten ausnützen kann. Im ersten Schritt werden wir über pgAdmin eine Datenbank aufsetzen und mit Daten aus dem IMDb Dataset[2] befüllen. Diese verwenden wir als Basis für die weiteren Schritte, wo wir verschiedene Abfragen ausführen und vergleichen.

a) *1 Punkt* Navigieren Sie zu `https://www.imdb.com/interfaces/` und verschaffen Sie sich einen Überblick über die Tabellen `title_akas` und `title_ratings` (Attribute, Schema, Datentypen, ...). Erstellen Sie über pgAdmin eine neue Datenbank namens `imdb` und bereiten Sie das Schema dieser Tabelle vor. Achten Sie darauf, korrekte Datentypen zu verwenden. Laden Sie anschließend das Paket `imdb-data.zip` (OLAT) herunter und entpacken Sie die tsv-Dateien (verwenden Sie die Dateien von OLAT!). Diese Dateien enthalten Daten für die Tabellen `title_akas` und `title_ratings`. Importieren Sie diese Daten in die entsprechende Tabelle. Beachten Sie, dass Sie das beim Import folgendes explizit angeben müssen (weitere Hinweise finden Sie weiter unten):

- NULL-Werte sind durch $\backslash N$ gekennzeichnet
- die Dateien enthalten einen Header

[2] `https://www.imdb.com/interfaces/`

- Spalten sind durch Tabs separiert
- ” ist das Quote-zeichen
- ’ müssen ”escaped” werden

Führen Sie schließlich folgende Abfrage aus und geben Sie das Ergebnis als csv-Datei ab.

```
1   SELECT COUNT(titleid)
2   FROM   title_akas
3   WHERE  language IS NULL
```

Abgabe

2a_create_db.sql

2a_create_table_akas.sql

2a_create_table_ratings.sql

2a_query_result.csv

Lösung

```
1   CREATE DATABASE imdb;
```

–

```
1   CREATE TABLE title_akas (
2    titleId VARCHAR(64),
3    ordering INTEGER,
4    title TEXT,
5    region VARCHAR(32),
6    language VARCHAR(32),
7    types TEXT,
8    attributes TEXT,
9    isOriginalTitle BOOLEAN
10  )
```

–

```
1   CREATE TABLE title_ratings (
2    tconst VARCHAR(32) PRIMARY KEY,
3    averageRating FLOAT,
4    numVotes INTEGER
5   )
```

–

```
1   "count"
2   "2098409"
```

Hinweis

Falls Sie pgadmin verwenden, können Sie die tsv-Datei importieren indem Sie die Import/Export Funktion verwenden (Rechtsklick auf die gewünschte Tabelle). Dort können Sie auch weitere Einstellungen zum Einlesen vornehmen.

Hinweis

Achten Sie darauf, dass die Reihenfolge der Spalten übereinstimmen. Wenn es zu Fehlermeldungen beim Importieren kommt, stellen Sie sicher, dass die Definition der Tabelle korrekt ist oder machen Sie die Datentypen weniger restriktiv.

Hinweis

Wenn Sie Docker verwenden, können Sie die tsv-Dateien in die imdb-Datenbank importieren, indem Sie folgenden Befehl ausführen:

```
cat <filename>.tsv | docker-compose exec -T db psql -U postgres \
 -d imdb -c "COPY <tablename>(<attr\_1>, ..., <attr\_n>) \
FROM STDIN CSV DELIMITER E'\t' NULL '\N' QUOTE '\"' ESCAPE '''' HEADER;"
```

Wobei Sie für `<filename>` den Pfad und den Namen der tsv-Datei angeben müssen, sowie für `<tablename>` den Tabellenname und für `<attr_i>` die entsprechenden Attribute angeben müssen.

b) *1 Punkt* Führen Sie nun folgenden Abfrage aus und notieren Sie die Zeit (in der Query History zu sehen). Schreiben Sie diese Zeit im Format `<seconds>s<milliseconds>msec` in die Abgabedatei (es gibt hier natürlich nicht *die eine* korrekte Ausführungszeit).

```
SELECT MAX(ordering)
FROM   title_akas
```

Führen Sie dieselbe Abfrage mit EXPLAIN[3] aus. Wie wird die Abfrage vom DB-System abgearbeitet? Beschreiben Sie was in den einzelnen Schritten des Plans passiert, indem Sie auf die Operationen eingehen (z.B Aggregate, Partial Aggregate, Finalize Aggregate, Gather, Seq Scan, Parallel Seq Scan, Merge . . .). Falls Sie pgAdmin verwenden, können Sie sich den Abfrageplan auch grafisch anzeigen lassen (F7).

Abgabe

- 2b_time.txt
- 2b_explain_result.txt
- 2b_explain_description.txt

[3] https://www.postgresql.org/docs/11/using-explain.html

Lösung ✔

```
1    0s959msec
-
1    "Finalize Aggregate (cost=228120.61..228120.62 rows=1 width=4)"
2    "  ->  Gather  (cost=228120.40..228120.61 rows=2 width=4)"
3    "        Workers Planned: 2"
4    "        ->  Partial Aggregate (cost=227120.40..227120.41 rows=1 width=4)"
5    "              ->  Parallel Seq Scan on title_akas
6    (cost=0.00..211151.92 rows=6387392 width=4)"
```

Es wird eine sequentielle Suche über die `title_akas` Relation gemacht. Diese wird parallelisiert und das Ergebnis der 2 Workers wird anschließend zusammengefasst (Gather) und auf diesem Ergebnis wird nochmals die Aggregatfunktion aufgerufen (beide Workers ermitteln den Maximalwert für ein Teil der Relation).

c) *1 Punkt* Erstellen Sie mittels `CREATE INDEX`[4] einen aufsteigend sortierten `BTREE`-Index auf die Spalte `ordering` in der `title_akas` Relation. Führen Sie folgende Abfrage aus und notieren Sie die Ausführungszeit im Format `<seconds>s<milliseconds>msec`.

```
1    SELECT MAX(ordering)
2    FROM   title_akas
```

Lassen Sie sich wieder mit `EXPLAIN` den vom Optimierer ermittelten Abfrageplans ausgeben und interpretieren Sie das Ergebnis indem Sie die einzelnen Schritte des Abfrageplans beschreiben. Erklären Sie dabei warum Begriffe wie *Limit, Forward, Backward, Scan, using* vorkommen, falls sie vorkommen. Beschreiben Sie wie sich das Erstellen des Index sich auf Ausführungszeit der Abfrage ausgewirkt hat. Führen Sie die Abfrage noch zwei weitere Male aus und achten Sie dabei auf die Ausführungszeit. Was können Sie beobachten? Führen Sie auch das in Ihrer Beschreibung an.

Abgabe ↑

- 2c_create_index.sql
- 2c_time.txt
- 2c_explain_result.txt
- 2c_explain_description.txt

Lösung ✔

```
1    CREATE INDEX ordering_index
2        ON public.title_akas USING btree
```

[4] https://www.postgresql.org/docs/11/sql-createindex.html

```
3       (ordering ASC NULLS LAST)
4       TABLESPACE pg_default;
-
1   0s45msec
-
1   "Result  (cost=0.46..0.47 rows=1 width=4)"
2   "  InitPlan 1 (returns $0)"
3   "    ->  Limit  (cost=0.43..0.46 rows=1 width=4)"
4   "          ->  Index Only Scan Backward using ordering_index on title_akas
    (cost=0.43..322717.68 rows=15451957 width=4)"
5   "                Index Cond: (ordering IS NOT NULL)"
```

Diese Abfrage verwendet den zuvor erstellten B-Baum-Index `ordering_index`. Das erkennt man im Output des `EXPLAIN` Operators daran: *"... Index Only Scan Backward using ordering_index ..."*. Dieser wird rückwärts traversiert (da dieser Index ja aufsteigend sortiert ist). Anschließend wird die Ergebnisrelation auf das erste Tupel limitiert, dieser wird als Ergebnis zurückgeliefert. Die Verwendung des Index wirkt sich positiv auf die Ausführungszeit dieser Abfrage aus (ca. um Faktor 8 schneller als ohne Verwendung des Index).

d) *1 Punkt* Man spricht von einem zusammengesetzten (compound oder composite) Index, wenn dieser mehrere Spalten beinhaltet. Damit werden im B-Baum nicht einzelne Werte, sondern Tupel indiziert (z.B. (region, titleid)). Erstellen Sie einen zusammengesetzten B-Baum-Index über die Spalten `region` und `titleid` – in dieser Reihenfolge und beide Attribute aufsteigend sortiert. Folgende Abfragen sind gegeben:

1) Eine Abfrage, die beide Spalten `region` und `titleid` einschränkt

```
1   SELECT *
2   FROM   title_akas
3   WHERE  titleid = 'tt6996876'
4   AND    region = 'DE';
```

2) Eine Abfrage, die nur `titleid` einschränkt

```
1   SELECT *
2   FROM   title_akas
3   WHERE  titleid = 'tt6996876';
```

3) Eine Abfrage, die nur `region` einschränkt

```
1   SELECT *
2   FROM   title_akas
3   WHERE  region = 'DE';
```

4) Eine Abfrage, die nach `region` und `titleid` aufsteigend sortiert

```
1   SELECT   *
2   FROM     title_akas
3   ORDER BY region ASC, titleid ASC
```

5) Eine Abfrage, die nach `region` aufsteigend und nach `titleid` absteigend sortiert

```
1   SELECT   *
2   FROM     title_akas
3   ORDER BY region ASC, titleid DESC
```

6) Eine Abfrage, die nach `region` und `titleid` absteigend sortiert

```
1   SELECT   *
2   FROM     title_akas
3   ORDER BY region DESC, titleid DESC
```

Betrachten Sie jeweils das Ergebnis des `EXPLAIN`-Aufrufs dazu. Welche Unterschiede in Ausführungszeit und Abarbeitungsstrategie bemerken Sie? Beschreiben Sie für welche Abfragen der zusammengesetzte Index Vorteile bringt und für welche nicht. Falls der zusammengesetzte Index bei einer Abfrage nicht verwendet wird, begründen Sie warum das (wahrscheinlich) so ist.

Abgabe

- 2d_create_compound_index.sql
- 2d_interpretation.txt

Lösung

```
1   CREATE INDEX compound_idx
2     ON public.title_akas USING btree
3     (region ASC NULLS LAST, titleid ASC NULLS LAST)
4     TABLESPACE pg_default;
-
```

Es ist stark vom Optimierer abhängig, wann sich ein DBS dazu entscheidet, einen Index zu verwenden und wann nicht. Hier spielt die Art der Indizes und Selektivität der Abfragen eine große Rolle. Werden viele Daten mit einer Abfrage selektiert, so ist meistens die sequentielle (evtl. parallel) effizienter (das Mitlesen von Indizes kostet auch). Werden wenig Daten selektiert, so entscheidet sich der Optimierer meistens dazu, einen Index zu verwenden, falls einer angelegt wurde. Diesen Unterschied merkt man bei den Abfragen 2 und 3. Wäre die Selektivität der Abfrage 3 größer, so würde sich Optimierer wahrscheinlich doch für die Verwendung des Indizes entscheiden. Für Abfrage 1 wird auch der zusammengesetzte Index verwendet da beide Attribute eingeschränkt werden.

Bei den Abfragen 4, 5 und 6 ist die Sortierreihenfolge ausschlaggebend dafür, ob der Index verwendet wird oder nicht. Wird nach den Attributen in aufsteigender/absteigender gefragt, so kann der B-Baum-Index vorwärts/rückwärts traversiert werden. Wird nach re-

gion aufsteigend und nach titleid absteigend sortiert, so ist der zusammengesetzte Index nutzlos.

	Zeit ohne Index	Zeit mit Index	Index verwendet
Abfrage 1	775ms	59ms	✓
Abfrage 2	801ms	846ms	✗
Abfrage 3	3s 352ms	2s 309 ms	✓
Abfrage 4	3min 12s	46s 379ms	✓
Abfrage 5	2min 11s	2min 18s	✗
Abfrage 6	2min 18s	37s 621ms	✓

e) *2 Punkte* Betrachten Sie die zwei folgenden Varianten einer SQL-Abfrage:

```sql
WITH languagecounts AS (
  SELECT    titleid, COUNT(language) AS no_languages
  FROM      title_akas
  GROUP BY  titleid
)
SELECT      tconst, no_languages
FROM        title_ratings
INNER JOIN  languagecounts
ON          title_ratings.tconst = languagecounts.titleid
WHERE       no_languages > (
  SELECT    AVG(no_languages)
  FROM      languagecounts
)
```

–

```sql
SELECT      tconst, COUNT(language) AS no_languages
FROM        title_akas
INNER JOIN  title_ratings
ON          title_ratings.tconst = title_akas.titleid
GROUP BY    tconst
HAVING      COUNT(language) > (
  SELECT    AVG(languagecount)
  FROM (
    SELECT    COUNT(language) AS languagecount
    FROM      title_akas
    GROUP BY  titleid
  ) AS tmp
)
```

Wir möchten diese Queries gerne näher analysieren und inspizieren, wie genau das DBMS diese Abfrage abarbeitet. Der JSON-Output von EXPLAIN gibt uns hierzu ausführliche Informationen, die wir im Folgenden vergleichen möchten. Beantworten Sie dazu folgende Fragen:

1) Welche Abfrage ist schneller?
2) Welche Schritte werden für die Abarbeitung der WITH-Abfrage durchgeführt? Erklären Sie die wichtigsten Operationen. Welche Schritte sind zeitlich gesehen die Bottlenecks der WITH-Abfrage?
3) Welche Schritte werden für die Abarbeitung der HAVING-Abfrage durchgeführt? Erklären Sie die wichtigsten Operationen. Welche Schritte sind zeitlich gesehen die Bottlenecks der HAVING-Abfrage?

Abgabe

`2e_answer_1.txt`

`2e_answer_2.txt`

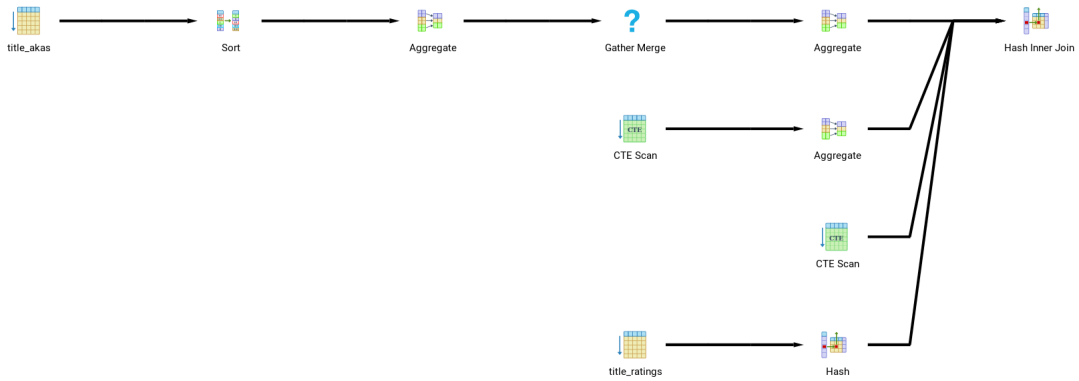
`2e_answer_3.txt`

Lösung

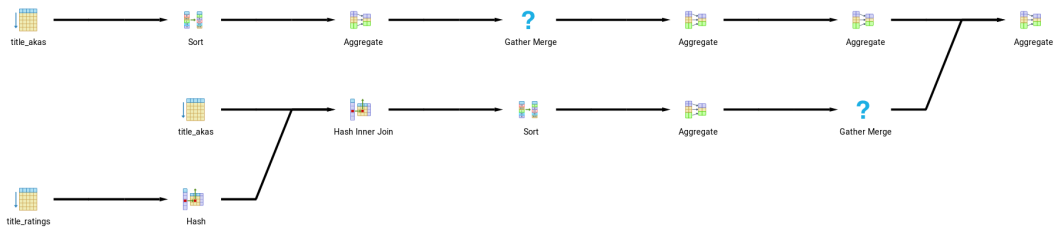
Die Abfrage mit WITH ist schneller. Das ist auch zu erwarten wenn man sich die geschätzten Kosten des `EXPLAIN`-Aufrufs für die beiden Abfragen anschaut. Wie die angegebenen Kosten zu interpretieren sind, wird hier erklärt: https://www.postgresql.org/docs/11/using-explain.html. Es ist eine Addition aus Kosten für das Lesen einer Seiten und CPU-Kosten pro Tupel. Die Schätzung kann hier natürlich auf verschienden Rechnern unterschiedlich groß sein, denn sie ist auch abhängig von der verwendeten Hardware.

- `Hash Join (cost=949737.33..977973.50 rows=182701 width=18)`. Bei der WITH-Abfrage werden die Kosten also auf 977973.50 geschätzt.
- `Finalize GroupAggregate (cost=1852719.98..2167990.03 rows=399051 width=18)`. Bei der HAVING-Abfrage werden die Kosten auf 2167990.03 geschätzt.

Bei der Abarbeitung der WITH-Abfrage wird zuerst eine CTE erstellt, die später mit der `title_ratings` Tabelle gejoint wird und auf Basis der die durchschnittliche Anzahl an verschiedenen Sprachen ermittelt wird.



Die HAVING-Abfrage joint zuerst die Relation `title_akas` mit `title_ratings`. Das Ergebnis dieses JOINS wird nach `tconst` gruppiert und es werden jene Tupel herausgefiltert, wo die Anzahl an Sprachen größer ist als der Durchschnitt.



Um die Bottlenecks der Abfragen zu bestimmen, schaut man sich die Operationen an, die die größten Anteil an den geschätzten Gesamtkosten haben. Die Kosten für das Zählen der Sprachen, nach `titleid` gruppiert, wird auf 895820.04 geschätzt (im EXPLAIN zu sehen bei `Finalize GroupAggregate (cost=756958.35..895820.04 rows=548103 width=18)`). Das macht bei der WITH-Abfrage umgerechnet $\approx 91.5\%$ der Gesamtkosten aus. Bei der HAVING-Abfrage sind es nur $\approx 25.2\%$, weil dort der Join von `title_akas` mit `title_ratings` der größere Bottleneck ist ($\approx 56.7\%$).

Wichtig: Laden Sie bitte Ihre Lösung in OLAT hoch und geben Sie mittels der Ankreuzliste auch unbedingt an, welche Aufgaben Sie gelöst haben. Die Deadline dafür läuft am Vortag des Proseminars um 23:59 (Mitternacht) ab.