Performance-Verbesserung auf der Datenbank Praktikum 5

1. Vorteile und Nachteile von Indizes
   Vorteil: Die Suche findet nicht mehr in linearer Laufzeit statt, sondern in logarithmischer Basis. Das kann eine enorme Performance Verbesserung für Select Abfragen sein.

   Nachteil: Der binäre Suchbaum muss gepflegt werden, weshalb **delete** und **insert** länger dauern. Das ist auch der Grund, warum es nicht sinnvoll ist auf jeder Tabelle einen Index zu verwenden, da dies potenziell sehr performance-belastend ist.

2. Prüfen Sie, für welche Tabellen und welche Spalte(n) es auf der Datenbank bereits einen Index gibt.

```
SELECT *
FROM pg_indexes
WHERE schemaname = 'public'
```

ta Output | Explain | Messages | Notifications

schemaname name	tablename name	indexname name	tablespace name	indexdef text
public	answer	answer_pkey	[null]	CREATE UNI...
public	category	category_pkey	[null]	CREATE UNI...
public	category	category_name...	[null]	CREATE UNI...
public	question	question_pkey	[null]	CREATE UNI...
public	player	player_pkey	[null]	CREATE UNI...
public	player	player_playerna...	[null]	CREATE UNI...
public	game	game_pkey	[null]	CREATE UNI...

3. Prüfen Sie, welchen Wert die Systemtabelle pg_stat_user_tables für die Anzahl der Datensätze (n_live_tup) in den Tabellen *PLAYER* und *GAME* und die *assoziierten Tabellen* enthält:

```
16  SELECT relname, n_live_tup
17  FROM pg_stat_user_tables
```

Data Output | Explain | Messages | Notifications

	relname name	n_live_tup bigint
1	player	10001
2	question	200
3	game	1000001
4	category	51
5	gamequesti...	14140479
6	answer	800

4. Generieren Sie nochmals die Explains für die Queries, die Sie in der Vorbereitung zu diesem Praktikum erstellt haben. Was hat sich geändert? Welche Kosten werden nun für die Ausführung einer Query angegeben?

5. Welche Spalten sind sinnvoll mit einem zusätzlichen Index zu belegen, um einen günstigeren Anfrageplan für die Verbund Abfragen zu erhalten? Implementieren Sie die entsprechenden Indexe:

   -> siehe bei Indizes: unter Update Explain 2:

Query2:

Startversion im Code 2:

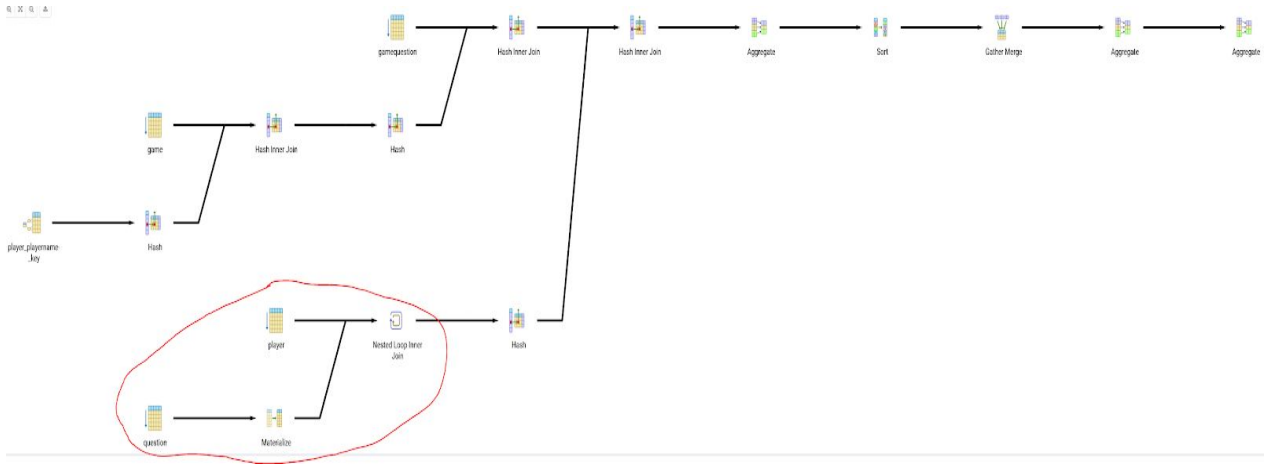
```
Select distinct g.gameid, g.startTime, count(KEY(q)),  "
+ "sum(case when VALUE(q) = True then 1 else 0 END) as
correctAnswerCount "
+ "from Game g join g.questionAnswer q, Player p "
+ "where g.player.playerName = :rndPlayer
```

Datenbank Umsetzung 2:

SELECT DISTINCT t0.GAMEID, t0.STARTTIME, COUNT(t1.QUESTIONID), SUM(CASE WHEN (t2.QUESTIONANSWER = true) THEN 1 ELSE 0 END) FROM PLAYER t4, PLAYER t3, gameQuestion t2, QUESTION t1, GAME t0 WHERE ((t4.PLAYERNAME = '7503') AND ((t4.PLAYERID = t0.player) AND ((t2.Game_GAMEID = t0.GAMEID) AND (t1.QUESTIONID = t2.questionAnswer_KEY)))) GROUP BY t0.GAMEID

Zeit: 3000 - 4000 ms

Explain2:



Hier ist uns der rot eingekreiste „inner join loop“ aufgefallen, welcher die Player-Tabelle ein zweites mal aufrief und die Zeilenanzahl explosionsartig erhöht hat (+ 2 Mio Zeilen). Dieser ließ sich auf einen Fehler in unserer Java-Implementation zurückführen: Player p

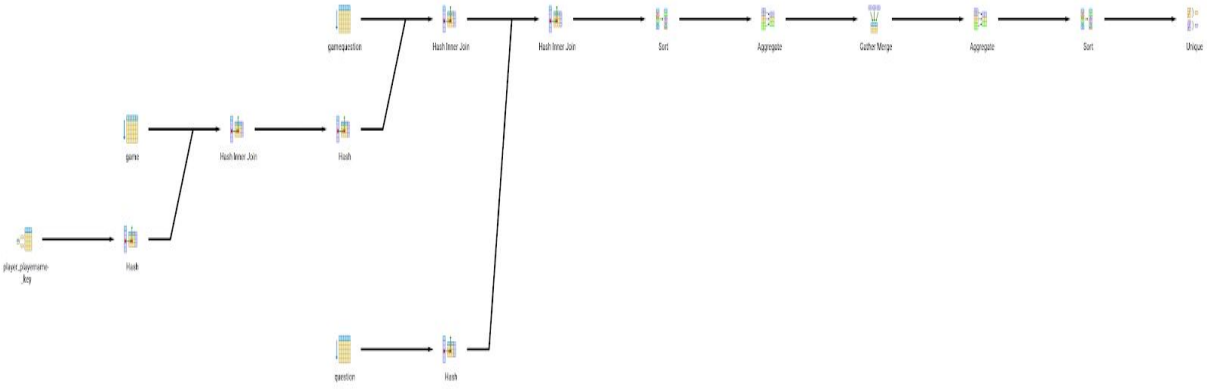
-> Dieser Eintrag war eigentlich unbenutzt und wurde daher einfach mal weggekürzt!

Neue Zeit: 800 – 1000ms

Update Datenbank Umsetzung 2:

```
SELECT DISTINCT t0.GAMEID, t0.STARTTIME, COUNT(t1.QUESTIONID), SUM(CASE
WHEN (t2.QUESTIONANSWER = true) THEN 1 ELSE 0 END) FROM PLAYER t3,
gameQuestion t2, QUESTION t1, GAME t0 WHERE ((t3.PLAYERNAME = '7503') AND
((t3.PLAYERID = t0.player) AND ((t2.Game_GAMEID = t0.GAMEID) AND (t1.QUESTIONID
= t2.questionAnswer_KEY)))) GROUP BY t0.GAMEID
```

Update Explain 2:



Indizes:

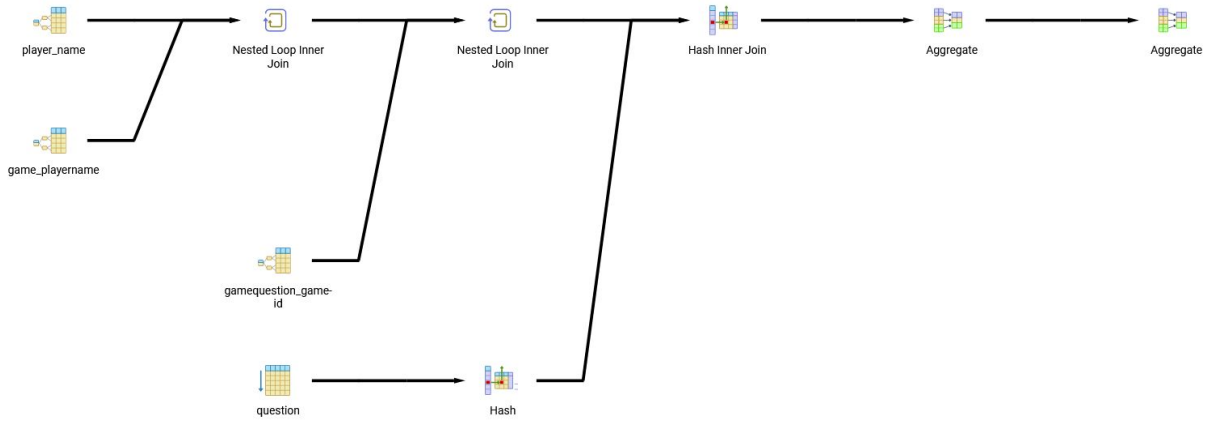
CREATE INDEX player_name ON player (playername)
- Verhindert das Suchen durch alle Spieler (1 Zeilen statt 10000)

CREATE INDEX game_playername ON game (player)
-Verhindert das Suchen durch alle Spiele (100 Zeilen statt 1000000)

CREATE INDEX gamequestion_gameid ON gamequestion (game_gameid)
-Verhindert das Suchen durch die Riesentabelle (14x100 Zeilen statt 3x4713182)

Neue Zeit: 5 – 40 ms (starke Tendenz zur 5)

Update Explain 2:



Query4:

Startversion im code 4:

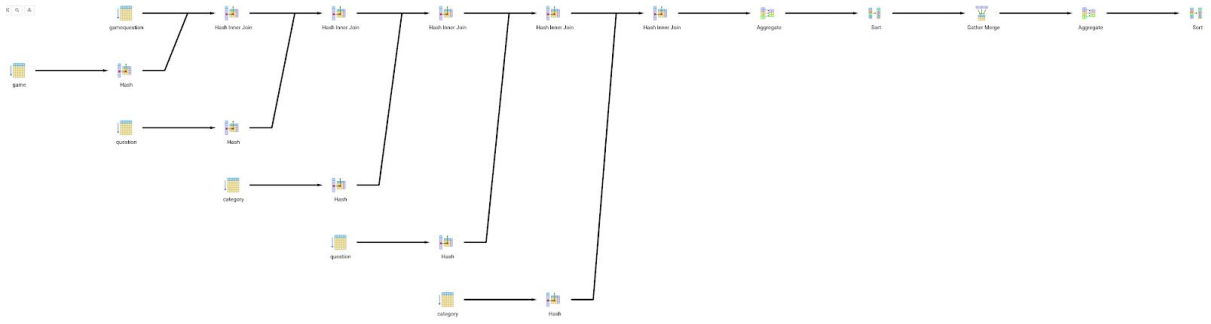
```
select c.name, c.categoryId, count (c) as cc "
+ "from Game g join g.questionAnswer qa, Question q, Category c where "
+ "q.questionId = Key(qa).questionId AND q.category.categoryId =
c.categoryId "
+ "group by c.categoryId order by cc desc
```

Datenbank Umsetzung 4:

```
SELECT t0.NAME, t0.CATEGORYID, COUNT(t0.CATEGORYID) FROM CATEGORY t5,
QUESTION t4, GAME t3, gameQuestion t2, QUESTION t1, CATEGORY t0 WHERE
(((t4.QUESTIONID = t1.QUESTIONID) AND (t5.CATEGORYID = t0.CATEGORYID)) AND
(((t2.Game_GAMEID = t3.GAMEID) AND (t1.QUESTIONID = t2.questionAnswer_KEY))
AND (t5.CATEGORYID = t4.CATEGORY_CATEGORYID))) GROUP BY t0.CATEGORYID
ORDER BY COUNT(t0.CATEGORYID) DESC
```

Zeit: 5000 - 10000 ms

Explain 4:



Hier zeigten sich eine ganze Reihe Fehler in unserer Java-Implementierung, Tabellen werden doppelt gejoint und die Datenmenge wurde ohne Grund ins unendliche aufgeblasen.

Nach etlichem Suchen unsere neue Lösung:

```
"select key(q).category as cat, count(cat) as c "
+ "from Game g join g.questionAnswer q "
+ "group by cat "
+ "order by c desc"
```

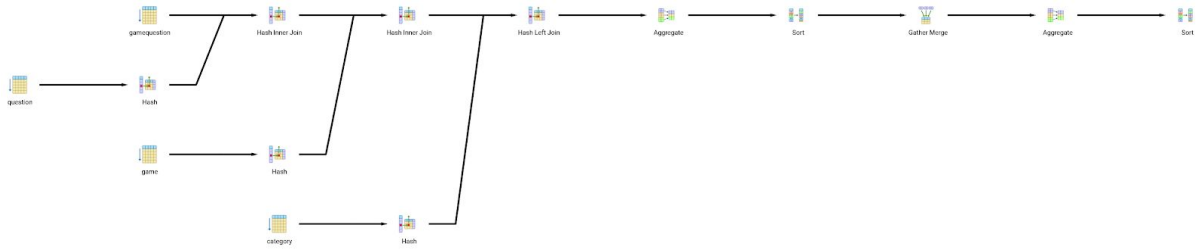
Update Datenbank Umsetzung 4:

SELECT t0.CATEGORYID, t0.NAME, COUNT(t0.CATEGORYID) FROM QUESTION t1 LEFT OUTER JOIN CATEGORY t0 ON (t0.CATEGORYID = t1.CATEGORY_CATEGORYID), GAME t3, gameQuestion t2 WHERE ((t2.Game_GAMEID = t3.GAMEID) AND (t1.QUESTIONID = t2.questionAnswer_KEY)) GROUP BY t0.CATEGORYID, t0.NAME ORDER BY COUNT(t0.CATEGORYID) DESC

Neue Zeit: 5000 – 8000 ms

Zwar weniger Joins, aber kein besonders großer Unterschied in der Performance, da die extra Tabellen recht klein waren.

Update Explain 4:



Indizes:

Leider helfen Indizes hier nicht, da alle Spieldaten gebraucht werden, um die Gesamtzahl genutzter Kategorien zu bekommen. Ein Index hilft beim Optimieren erst, wenn man nur auf Teile einer Tabelle zugreifen möchte (bzw. scheint der Optimizer Indexe jeglicher Sorte hier nicht für nötig zu halten).

Query1:

Startversion im code 1:

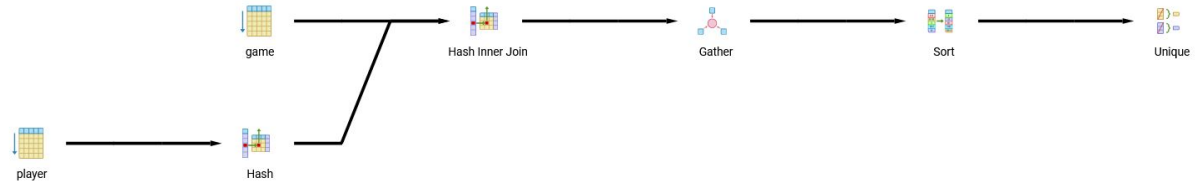
```
"select distinct g.player.playerName from Game g "
+ "where g.startTime between :start and :end order by g.gameid"
```

Datenbank Umsetzung 2:

```
SELECT DISTINCT t0.PLAYERNAME, t1.GAMEID FROM PLAYER t0, GAME t1 WHERE
((t1.STARTTIME BETWEEN '2020-01-01 00:00:00.0' AND '2020-01-01 23:59:59.0') AND
(t0.PLAYERID = t1.player)) ORDER BY t1.GAMEID
```

Zeit: 150 - 200 ms

Explain 1:



Ziel war es sowohl die große game Tabelle als auch die Spieler per Index zu durchsuchen, jedoch war es uns nur möglich die Spiele zu optimieren, da er darauf besteht die Spieler vorher abfragen \(^^)/

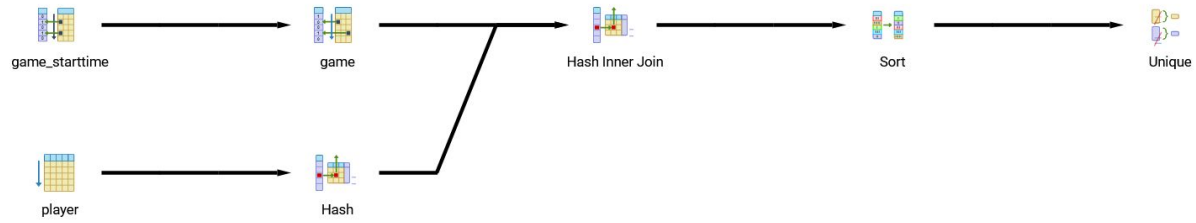
Indizes:

```
CREATE INDEX game_starttime ON game (starttime)
```

-> Immerhin reduziert sich die Anzahl an Games von 1 Mio auf (in unserem Testfall) 34 300.

Neue Zeit: ca 50 ms

Update Explain 1:



Spannende Beobachtung der Optimizer:

(lokale Datenbank)

- 1 - 12 Tage Zeitraum = Bitmap-Heap-Scan (mit Index)
- ab 13+ sequentiell (ohne Nutzung des Index)

(fbi-Datenbank)

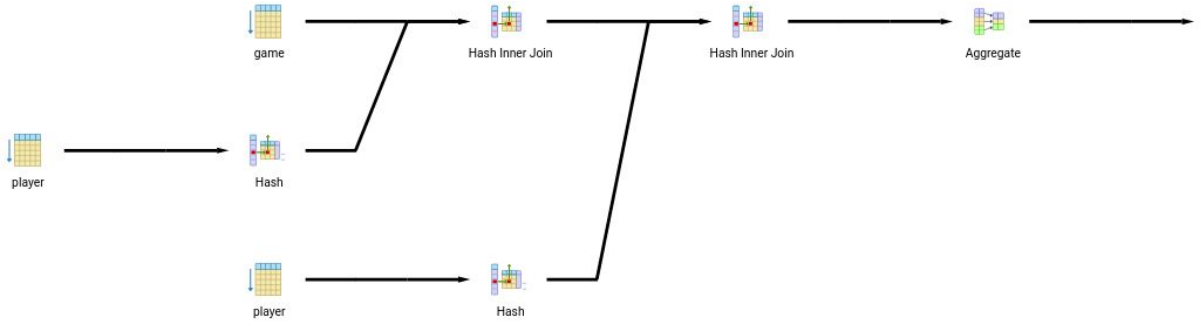
- 1 - 3 Tage Zeitraum Bitmap-Heap-Scan (mit Index)
- 4 – 18 Index-Scan (mit Index)
- 19+ sequentiell (ohne nutzung des Index)

Braucht er also mehr als x% Zeilen einer Tabelle scheint er sequenziell allen anderen Optionen vorzuziehen.

Optimierung der 3. Query

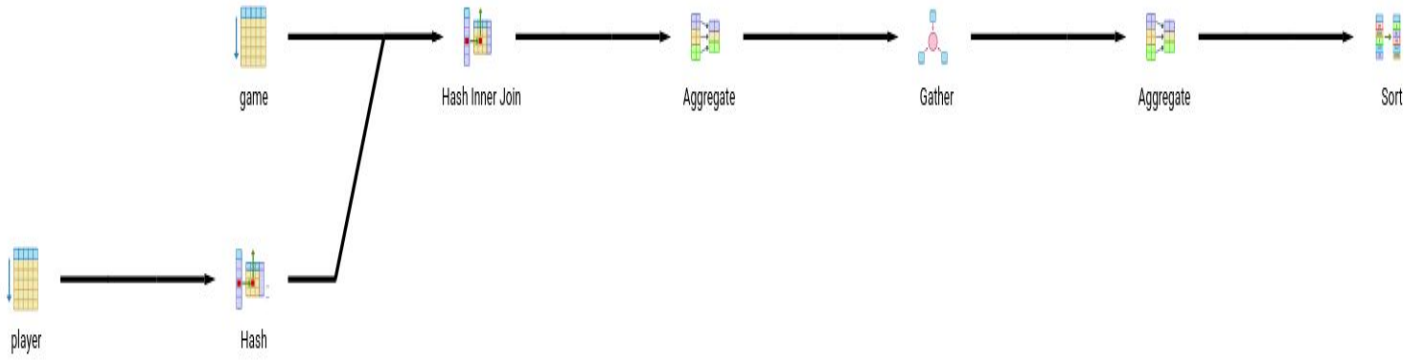
- **von:**

```
SELECT t0.PLAYERNAME, COUNT(t1.GAMEID) FROM PLAYER t0,
PLAYER t2, GAME t1
WHERE ((t0.PLAYERID = t2.PLAYERID) AND (t2.PLAYERID = t1.player))
GROUP BY t0.PLAYERNAME ORDER BY COUNT(t1.GAMEID) DESC
"select p.playerName, count(g) as gamecount from Player
p, Game g "
+ "where p.playerid = g.player.playerid group by
p.playerName order by gamecount desc"
```



- **zu:**

```
SELECT t0.PLAYERNAME, COUNT(t1.GAMEID)
FROM PLAYER t0, GAME t1
WHERE (t0.PLAYERID = t1.player)
GROUP BY t0.PLAYERNAME ORDER BY COUNT(t1.GAMEID) DESC
"select g.player.playerName, count(g) as gamecount from
Game g "
+ "group by g.player.playerName order by gamecount desc"
```


game
Hash Inner Join
Aggregate
Gather
Aggregate
Sort
player
Hash