11 Indexe effektiv einsetzen

Indexe sind essenziell für die Performance eines Datenbankschemas und der angeschlossenen Applikation. Im Vergleich zu anderen Datenbanksystemen bietet PostgreSQL eine große Bandbreite von verschiedenen Indextypen. Ein Grund dafür ist, dass Indexe konsequent von der Open Source Community auf der Basis von Anwenderanfragen mit innovativen Ideen entwickelt wurden. Darüber hinaus bietet PostgreSQL durch seine offene Architektur die Möglichkeit, zusätzliche Indextypen einfach selbst zu implementieren. Wie das geht, wird in diesem Kapitel an einem Beispiel dargestellt.

PostgreSQL stellt die folgenden Indextypen zur Verfügung:

- B-Tree-Index
- Block Range-Index (BRIN)
- Hash-Index
- Generalized Inverted Index (GIN)
- Generalized Search Tree-Index (GiST)
- Expression-Index
- Partieller Index
- Individueller Index

Indexe unterscheiden sich stark in Zweck und Technologie. Sie haben jedoch gemeinsam, dass sie in ihrer Struktur den Tuple Identifier (TID) verwenden, um auf die Daten der Tabelle zuzugreifen. Zur Erinnerung, ein TID besteht aus der Datei- und Blocknummer sowie der Position des Datensatzes innerhalb des Blocks und ist die schnellste Methode, um auf einen Datensatz gezielt zuzugreifen.

Die Syntax für das Erstellen eines Index lautet:

```
CREATE INDEX <index-name> ON <tabelle> USING <index-typ> (<spalte>);
```

Mit dem Plan, einen Index zu erstellen, sollte man sich darüber im Klaren sein, dass ein Index gepflegt werden muss. Operationen wie INSERT, UPDATE oder DELETE verlangsamen sich etwas, da der Index gepflegt werden muss. Für Tabellen mit wenigen Indexen ist dies in der Regel zu vernachlässigen. Für Tabellen, die eine große Anzahl von Indexen besitzen und auf denen viele DML-Operationen stattfinden, kann sich die Performance deutlich verschlechtern. Mit der Abfrage in Listing 11.1 kann festgestellt werden, wie häufig Indexe verwendet werden. Nicht oder wenig verwendete Indexe sollten gelöscht werden.

Listing 11.1 Die Verwendung von Indexen abfragen

```
(postgres@localhost:5432)[hanser]> SELECT t.tablename,
> c.reltuples::bigint AS num_rows,
> ps.indexrelname AS index_name,
> ps.idx_scan AS num_scans,
> ps.idx_tup_read AS t_read,
> ps.idx_tup_fetch AS t_fetched
> FROM pg_tables t
> LEFT JOIN pg_class c ON t.tablename = c.relname
> LEFT JOIN pg_index i ON c.oid = i.indrelid
> LEFT JOIN pg_stat_all_indexes ps ON i.indexrelid = ps.indexrelid
> WHERE t.schemaname NOT IN ('pg_catalog', 'information_schema')
> ORDER BY 1, 2;
 tablename | num_rows |   index_name    | num_scans | t_read | t_fetched
-----------+----------+-----------------+-----------+--------+-----------
 big       | 20000000 | big_hash_index  |         1 |   3192 |       3192
 big       | 20000000 | big_btree_index |         0 |      0 |          0
```

Indexe dienen dem optimierten Zugriff auf Daten und werden deshalb in PostgreSQL auch als „Access Methods“ (Zugriffsmethoden) bezeichnet. Sie finden diese in der View *pg_am* wieder.

Listing 11.2 Zugriffsmethoden in PostgreSQL

```
(postgres@localhost:5432)[hanser]> SELECT * FROM pg_am;
 oid  | amname |      amhandler       | amtype
------+--------+----------------------+--------
    2 | heap   | heap_tableam_handler | t
  403 | btree  | bthandler            | i
  405 | hash   | hashhandler          | i
  783 | gist   | gisthandler          | i
 2742 | gin    | ginhandler           | i
 4000 | spgist | spghandler           | i
 3580 | brin   | brinhandler          | i
```

Nicht alle Indextypen sind gleichermaßen für alle Vergleichsoperationen und Datentypen geeignet. Auch dies lässt sich im PostgreSQL-Katalog abfragen. Im Beispiel in Listing 11.3 werden alle Operationen, die für Arrays mit einem B-Tree-Index definiert sind, aufgelistet.

Listing 11.3 Vergleichsoperationen für einen Indextyp auflisten

```
postgres@localhost:5432)[hanser]> SELECT op.amopopr::regoperator
> FROM pg_opclass c, pg_opfamily f, pg_am am, pg_amop op
> WHERE c.opcname = 'array_ops'
> AND f.oid = c.opcfamily
> AND am.oid = f.opfmethod
> AND op.amopfamily = c.opcfamily
> AND am.amname = 'btree'
> AND op.amoplefttype = c.opcintype;
        amopopr
-----------------------
 <(anyarray,anyarray)
 <=(anyarray,anyarray)
 =(anyarray,anyarray)
 >=(anyarray,anyarray)
 >(anyarray,anyarray)
```

Mit dem Befehl REINDEX kann der Index neu aufgebaut werden.

Listing 11.4 Einen Index mit REINDEX neu aufbauen

```
(postgres@localhost:5432)[hanser]> REINDEX (VERBOSE) INDEX big_btree_index;
INFO:  index "big_btree_index" was reindexed
DETAIL:  CPU: user: 12.01 s, system: 3.39 s, elapsed: 96.21 s
REINDEX
```

Während der Index neu aufgebaut wird, werden Locks an der Tabelle aktiviert und DML-Operationen warten auf das Ende des Prozesses. Um Sperren zu verhindern, kann die Option CONCURRENTLY verwendet werden. PostgreSQL geht dann wie folgt vor:

1. Ein neuer temporärer Index wird zum Datenbankkatalog hinzugefügt (*pg_index*).
2. Der neue Index wird erstellt und der Marker *pg_index.indisready* wird auf „true“ gesetzt.
3. In der zweiten Phase werden die Sätze hinzugefügt, die während der ersten Phase in der Tabelle aktualisiert wurden.
4. Alle Constraints werden auf die neue Indexdefinition verwiesen und der Marker *pg_index.indisvalid* wird auf „true“ gesetzt.
5. Gleichzeitig wird für den alten Index der Wert *pg_index.indisready* auf „false“ gesetzt. Danach wird der alte Index gelöscht.

Nach dem Starten des Prozesses erscheint ein neuer Indexname, der während des Indexaufbaus für andere Sessions und den Optimizer nicht sichtbar ist:

```
(postgres@localhost:5432)[hanser]> SELECT c.relname, i.indisvalid, i.indisready
FROM pg_class c, pg_index i
WHERE i.indexrelid = c.oid
AND c.relname like 'big%';
        relname        | indisvalid | indisready
-----------------------+------------+------------
 big_btree_index       | t          | t
 big_btree_index_ccnew | f          | f
```

Ist der Switch abgeschlossen, wird der neue Index unter dem alten Namen wieder sichtbar gemacht:

```
(postgres@localhost:5432)[hanser]> SELECT c.relname, i.indisvalid, i.indisready
FROM pg_class c, pg_index i
WHERE i.indexrelid = c.oid
AND c.relname like 'big%';
        relname        | indisvalid | indisready
-----------------------+------------+------------
 big_btree_index       | t          | t
```

TIPP: Beachten Sie, dass ein normales VACUUM die Größe des Index nicht beeinflusst. Ein VACCUM FULL dagegen baut den Index neu auf und reduziert gegebenenfalls auch dessen Größe. Mit dem Befehl REINDEX wird die Größe des Index ebenfalls angepasst, wogegen die Größe der Tabelle unverändert bleibt.

Die Indexe unterscheiden sich auf Grund ihrer unterschiedlichen Architektur im Locking-Verhalten. B-Tree- und GIST-Indexe führen kurze Locks auf Page-Ebene aus. Die Sperren werden aufgehoben, sobald die INSERT-/UPDATE-/DELETE-Operation abgeschlossen ist. Sie stellen das bessere Konkurrenzverhalten, verglichen mit anderen Indextypen, zur Verfügung.

Ein Hash-Index bewirkt Locks auf Ebene eines Hash-Bucket. Auch hier werden die Sperren unmittelbar nach Ende der DML-Operation aufgehoben. Es kann zu mehr Konkurrenz kommen als bei B-Tree-Indexen und Deadlock-Situationen sind möglich.

Für GIN-Indexe gilt, dass, wie beim B-Tree-Index, Sperren auf Page-Ebene verwendet und sofort nach der Operation entfernt werden. Änderungen im Index sind jedoch aufwendiger. Pro Datensatz müssen in der Regel mehrere Änderungen im Index ausgeführt werden. Dies führt dazu, dass die Sperren länger gehalten werden und die Konkurrenzsituation schlechter als beim B-Tree-Index ist.

11.1 B-Tree-Index

Der B-Tree-Index ist der Klassiker und bei allen relationalen Datenbanksystemen anzutreffen. Wie der Name sagt, basiert er auf einer Baumstruktur und hat den Vorteil, dass er auf nahezu alle Spaltentypen einschließlich NULL-Werte anwendbar ist. Er ist sowohl für Gleichheits- als auch Bereichsbedingungen ein effizientes Werkzeug. Auch für die Sortierung von Daten sind B-Tree-Indexe sehr hilfreich.

„B-Tree“ ist der Standardindextyp in PostgreSQL. Wenn Sie einen Befehl CREATE INDEX ohne zusätzliche Angabe des Index-Typs verwenden, wird ein B-Tree-Index angelegt. Der Index realisiert eine Baumstruktur und ist für viele Anwendungsfälle gut geeignet. Wenn Sie nicht sicher sind bei der Wahl des Indextyps, können Sie mit einem B-Tree-Index generell nichts falsch machen und eine Performance-Verbesserung erwarten, auch wenn es möglicherweise einen besser geeigneten Indextyp gibt.

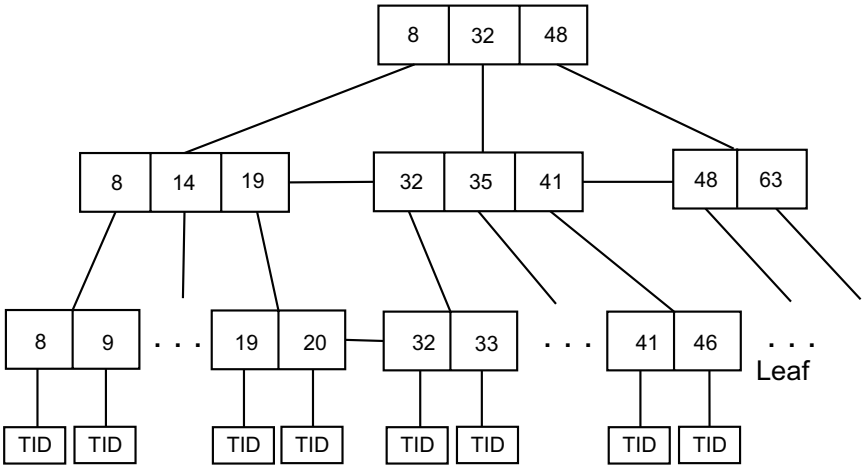


Bild 11.1 Struktur eines B-Tree-Index

Für B-Tree-Indexe gibt es eine Reihe von Annahmen, die für die PostgreSQL-Implementierung als erfüllt angesehen werden können. So sollten die Leaf-Knoten dieselbe Anzahl von internen Pages enthalten. Damit wird garantiert, dass die Suche nach einem beliebigen Wert ungefähr dieselbe Zeit in Anspruch nimmt. Die Tiefe eines Index (Anzahl der Verzweigungsebenen) sollte nicht zu groß sein. Man kann für PostgreSQL mit einer Tiefe von 4 bis 5 rechnen. Weiterhin sollten die Daten des Index sortiert und die Pages auf einer Ebene miteinander verbunden sein.

In einer Leaf Page befinden sich mehrere Indexeinträge, die über die TID auf den Datensatz verweisen.

Kann ein Eintrag nicht mehr untergebracht werden, wird eine neue Leaf Page erstellt. Eine Page wird dann geteilt und die entsprechenden Einträge auf den höheren Ebenen werden aktualisiert. Die Teilung einer Page kann auch die Root Page betreffen. In diesem Fall bekommt der Index eine zusätzliche Ebene.

B-Tree-Indexe verfügen nicht über einen eigenen Multi-Version-Control-Mechanismus. So können die verschiedenen Zeilenversionen eines Satzes jeweils eigene Indexeinträge besitzen. Nach einem COMMIT werden alte Zeilenversionen nicht mehr benötigt. Ein Prozess, der sich „Bottom-up Index Deletion“ nennt, sorgt dafür, dass alte, nicht mehr benötigte Indexeinträge entfernt werden.

Doppelte oder mehrfache Indexeinträge für denselben Spaltenwert sind relativ normal und entstehen während der Datenmanipulation. Für die Zusammenfassung der Einträge ist der Deduplizierungsprozess verantwortlich. Er läuft periodisch, jedoch nicht mit hoher Priorität. Er wird auch ausgelöst, wenn ein neuer Eintrag keinen Platz in einer Leaf Page finden kann. Die Indexgröße wird damit verringert.

TIPP: Eine vollständige Defragmentierung eines B-Tree-Index ist nur mit Hilfe eines Neuaufbaus (REINDEX) möglich. Für Tabellen, die häufigen Änderungen auf den indizierten Spalten unterliegen, ist ein Neuaufbau in regelmäßigen Abständen zu empfehlen. Indexzugriffe werden schneller und die Indexgröße wird reduziert. ■

DML-Operationen ziehen Aufwände in der Änderung der Indexstruktur nach sich. Das erklärt die Verlangsamung von DML-Operationen mit angeschlossenen Indexen.

B-Tree-Indexe können als eindeutige Indexe (UNIQUE) deklariert und für Primary Keys eingesetzt werden.

Beim Zugriff auf die Datensätze über den Index sind zusätzliche Leseoperationen für den Index erforderlich. Dennoch ist der Zugriff wesentlich schneller und weniger ressourcenintensiv im Vergleich zu einem Table Scan, solange die Anzahl der zu selektierenden Sätze vergleichsweise gering ist. Im Idealfall ist es nur ein Satz.

Listing 11.5 Zugriff auf die Tabelle über einen B-Tree-Index

```
(postgres@192.168.178.39:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM orderlines WHERE orderid = 4711;
                                    QUERY PLAN
-------------------------------------------------------------------------------

 Index Scan using ix_orderlines_orderid on orderlines  (cost=0.29..15.38 rows=5
width=18) (actual time=0.019..0.019 rows=2 loops=1)
   Index Cond: (orderid = 4711)
```

Ist die Anzahl der Sätze sehr hoch, wird der Zugriff über den B-Tree-Index ineffizient. PostgreSQL hält dafür zwei Alternativen bereit:

- Full Table Scan
- Bitmap Scan

Ein Full Table Scan wird gewählt, wenn die überwiegende Mehrheit der Sätze einer Tabelle selektiert werden müssen. Dabei wird ein Filter in den Scan eingebunden und die nicht relevanten Sätze herausgefiltert. Im Beispiel in Listing 11.6 werden fast alle Sätze der Tabelle gelesen. Über einen zusätzlichen Filter werden die nicht relevanten Sätze herausgefiltert.

Listing 11.6 Zugriff auf die Tabelle mit Full Table Scan

```
(postgres@192.168.178.39:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM orderlines WHERE orderid > 10;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on orderlines  (cost=0.00..1139.38 rows=60303 width=18) (actual
time=0.013..5.540 rows=60286 loops=1)
   Filter: (orderid > 10)
   Rows Removed by Filter: 64
```

Im Fall, dass eine größere Anzahl von Sätzen selektiert wird, jedoch nicht der Hauptteil der Tabelle, verwendet PostgreSQL alternativ einen Bitmap Scan. Ein Bitmap Scan kombiniert den Indexscan mit einem sequenziellen Scan und versucht so, die Vorteile des Indexscans zu nutzen, ohne die Nachteile des Zugriffs auf die Tabelle über den Index in Kauf zu nehmen.

Beim herkömmlichen Zugriff über den Index muss für jeden Indexeintrag die zugehörige Heap Page der Tabelle gelesen werden, was eine hohe Anzahl an Single-Block-Lesezugriffe zur Folge hat. Der Bitmap Scan verwendet die Vorteile des Indexzugriffs, ohne die Nachteile in Kauf nehmen zu müssen. Er funktioniert folgendermaßen:

1. Zuerst wird der Index gelesen und es wird eine Bitmap erstellt. Dabei wird ein Hashwert der Page-Nummern zusammen mit dem Offset der Pages erstellt. Dieser Teil ist im Ausführungsplan als „Bitmap Index Scan“ sichtbar.
2. Im zweiten Schritt wird der Bitmap durchgegangen und auf die Pages der Tabelle zugegriffen (im Ausführungsplan als „Bitmap Heap Scan“ sichtbar).

Listing 11.7 Zugriff auf die Tabelle mit einem Bitmap Scan

```
(postgres@192.168.178.39:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM orderlines WHERE orderid > 11000;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on orderlines  (cost=93.90..539.56 rows=4853 width=18) (actual
time=0.258..0.564 rows=4951 loops=1)
   Recheck Cond: (orderid > 11000)
   Heap Blocks: exact=33
   ->  Bitmap Index Scan on ix_orderlines_orderid  (cost=0.00..92.69 rows=4853
width=0) (actual time=0.251..0.251 rows=4951 loops=1)
         Index Cond: (orderid > 11000)
```

Dieser Mechanismus ist schneller als ein Indexscan mit Zugriff auf jeden einzelnen Datensatz, da die Daten der Tabelle in ihrer Reihenfolge, so wie bei einem sequenziellen Scan,

gelesen werden können. Ein Zugriff auf jeden einzelnen Datensatz über den Index bedeutet dagegen viele kleine Lesezugriffe (Random I/O), die bei größeren Datenmengen ineffektiv und langsam werden können. Eine sehr intelligente Zugriffsmethode, die andere professionelle Datenbanksysteme vermissen lassen.

In Kapitel 4 haben Sie bereits die Extension *pageinspect* kennengelernt, um interne Inhalte und Strukturen einer Tabelle anzuzeigen. Die Extension hält auch Funktionen für Indexe bereit. Erstellen Sie die Erweiterung, falls noch nicht geschehen:

```
(postgres@localhost:5432)[hanser]> CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

Für einen B-Tree-Index können mit der Funktion *bt_page_stats* Informationen einer einzelnen Page eines Index angezeigt werden.

Listing 11.8 Statistische Informationen einer Index Page anzeigen

```
(postgres@localhost:5432)[hanser]> SELECT * FROM bt_page_stats('ix_cust_username', 1);
 blkno | type | live_items | dead_items | avg_item_size | page_size | free_size | …
-------+------+------------+------------+---------------+-----------+-----------+-----
     1 | l    |        264 |          0 |            23 |      8192 |       796 | …
```

Die Inhalte einer Index Page lassen sich mit der Funktion *bt_page_items* auslesen. Hier findet man neben Informationen über Größe und Status der Indexeinträge die TID und den Spaltenwert im Hex-Format.

Listing 11.9 Den Inhalt einer B-Tree-Page auslesen

```
(postgres@localhost:5432)[hanser]>  SELECT itemoffset, ctid, itemlen, nulls,
vars, data, dead, htid
> FROM bt_page_items('ix_cust_username', 11);
 itemoffset |   ctid   | itemlen | nulls | vars |        data        | dead |   htid
------------+----------+---------+-------+------+--------------------+------+---------
          1 | (301,1)  |      24 | f     | t    | 15  …  00 00 00  |      |
          2 | (295,25) |      24 | f     | t    | 15  …  00 00 00  | f    | (295,25)
          3 | (295,26) |      24 | f     | t    | 15  …  00 00 00  | f    | (295,26)
          4 | (295,27) |      24 | f     | t    | 15  …  00 00 00  | f    | (295,27)
          5 | (295,28) |      24 | f     | t    | 15  …  00 00 00  | f    |
(295,28)
```

Mit der Abfrage in Listing 11.10 lässt sich ermitteln, welche B-Tree-Indexe einer hohen Fragmentierung unterliegen und Kandidaten für einen Neuaufbau sind.

Listing 11.10 Abfrage zur Ermittlung des Grads der Fragmentierung

```
(postgres@localhost:5432)[hanser]> SELECT i.indexrelid::regclass,
> s.leaf_fragmentation
> FROM pg_index AS i
>  JOIN pg_class AS t ON i.indexrelid = t.oid
>  JOIN pg_opclass AS opc ON i.indclass[0] = opc.oid
>  JOIN pg_am ON opc.opcmethod = pg_am.oid
>  CROSS JOIN LATERAL pgstatindex(i.indexrelid) AS s
> WHERE t.relkind = 'i'
>   AND pg_am.amname = 'btree';
                 indexrelid                  | leaf_fragmentation
---------------------------------------------+--------------------
```

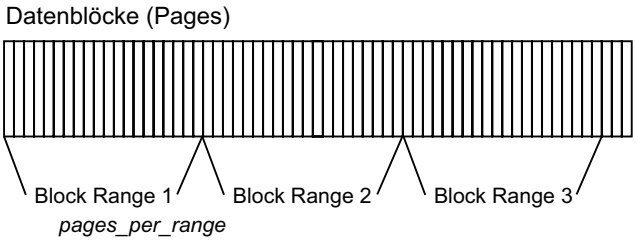
```
  pg_toast.pg_toast_1255_index                |                0
  pg_proc_proname_args_nsp_index              |             9.38
  pg_type_typname_nsp_index                   |               50
  pg_attribute_relid_attnam_index             |                0
  pg_class_relname_nsp_index                  |               25
  pg_class_tblspc_relfilenode_index           |            33.33
. . .
```

11.2 Block-Range-Index (BRIN)

B-Tree-Indexe sind effizient und universell einsetzbar mit vielen möglichen Operatoren. Ein Nachteil ist jedoch, dass sie nahezu linear mit der Größe der Tabelle wachsen. Große Indexe erhöhen die Anzahl der Leseoperationen von der Disk bei SQL-Abfragen. Eine tiefe Verzweigung macht Indexscans langsamer.

Für sehr große Tabellen ist es ratsam, nach Alternativen zum B-Tree-Index Ausschau zu halten. Seit der Version 9.5 gibt es Block-Range-Indexe. Dabei werden in einem Block nicht einzelne Spaltenwerte gespeichert, sondern Datenblöcke indiziert. Für jeden Block werden ein Minimal- und ein Maximalwert in Form von Bitmaps gespeichert (siehe Bild 11.2).

Das Attribut *pages_per_range* legt fest, wie viele Datenblöcke in einer Block Range zusammengefasst werden. Die Festlegung erfolgt mit der Erstellung des Indexes. Je größer der Wert, desto kleiner wird der Index. Wird der Wert zu groß gewählt, müssen bei Abfragen möglicherweise mehr Blöcke gescannt werden. Der „Query Executer" muss schließlich noch die Datensätze entfernen, die sich zwar in der Block Range befinden, jedoch der WHERE-Bedingung der SQL-Anweisung nicht genügen. Der Wartungsaufwand steigt ebenfalls mit der Anzahl der Blöcke. Der Standardwert für *pages_per_range* ist 128. Das Attribut *autosummarize* legt fest, ob eine automatische Pflege der Summenwerte erfolgen soll.



Block	Minimum	Maximum
1	2018-01-02	2018-01-06
2	2018-01-04	2018-01-12
3	2018-01-10	2018-01-22

Bild 11.2 Die Architektur eines Block-Range-Index

Ein BRIN-Index kann einen Performance-Vorteil gegenüber einem B-Tree-Index bringen. Sein entscheidender Vorteil liegt in der geringen Größe. Gerade für sehr große Tabellen ist der Unterschied signifikant. Der Aufbau des Index macht klar, wann er seine Stärken entfalten kann. Die größten Effekte werden erzielt, wenn die Werte der indizierten Spalte weitgehend in Sortierreihenfolge vorliegen und möglichst wenigen Änderungen unterliegen. Ein klassisches Beispiel ist das Eingangsdatum einer Auftragstabelle. Neue Aufträge werden in zeitlicher Reihenfolge aufgenommen und für die gespeicherten Sätze ändert sich der Wert nicht.

Für das folgende Beispiel wird in Listing 11.11 eine Tabelle mit mehr als 30 Millionen Sätzen angelegt.

Listing 11.11 Eine große Tabelle anlegen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE temperature(
> location_id        INTEGER,
> t_time             TIMESTAMP,
> t_celsius          INTEGER);
CREATE TABLE
(postgres@localhost:5432)[hanser]> INSERT INTO temperature
> VALUES(1, generate_series('2020-01-01'::timestamp,'2020-12-31'::timestamp,
'1 second'), round(random()*100)::int);
INSERT 0 31536001
```

Abfragen erfolgen nach der Spalte *t_time*, in der sich Daten vom Typ TIMESTAMP befinden. Zunächst wird ein B-Tree-Index auf die Spalte gelegt (siehe Listing 11.12).

Listing 11.12 Einen B-Tree-Index erstellen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_temp_btree
> ON temperature (t_time);
CREATE INDEX
Time: 59181,418 ms (00:59,181)
(postgres@localhost:5432)[hanser]> \d temperature
                        Table "public.temperature"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 location_id | integer                     |           |          |
 t_time      | timestamp without time zone |           |          |
 t_celsius   | integer                     |           |          |
Indexes:
    "i_temp_btree" btree (t_time)
```

Der Ausführungsplan für eine Abfrage zeige einen parallelen Indexscan auf den B-Tree-Index (siehe Listing 11.13).

Listing 11.13 Paralleler Indexscan mit B-Tree-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT AVG(t_celsius) FROM temperature
> WHERE t_time > '2020-02-28' AND t_time < '2020-04-01';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=86171.40..86171.41 rows=1 width=32) (actual
time=327.681..337.400 rows=1 loops=1)
   ->  Gather  (cost=86170.88..86171.38 rows=5 width=32) (actual
time=303.596..313.503 rows=6 loops=1)
```

```
         Workers Planned: 5
         Workers Launched: 5
         ->  Partial Aggregate  (cost=85170.88..85170.88 rows=1 width=32) (actual
time=244.592..244.593 rows=1 loops=6)
               ->  Parallel Index Scan using i_temp_btree on temperature
(cost=0.56..83743.74 rows=570853 width=4) (actual time=0.248..188.736 rows=475200
loops=6)
                     Index Cond: ((t_time > '2020-02-28 00:00:00'::timestamp without
time zone) AND (t_time < '2020-04-01 00:00:00'::timestamp without time zone))
 Planning Time: 2.819 ms
 Execution Time: 341.319 ms
(postgres@localhost:5432)[hanser]> SELECT AVG(t_celsius) FROM temperature
> WHERE t_time > '2020-02-28' AND t_time < '2020-04-01';
         avg
---------------------
 50.0099456404130333
(1 row)
Time: 748,957 ms
```

Für den weiteren Test wird ein BRIN-Index mit einer Block Range von „128“ angelegt (siehe Listing 11.14).

Listing 11.14 Einen Block-Range-Index anlegen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_temp_brin
> ON temperature USING BRIN (t_time) WITH (autosummarize);
CREATE INDEX
```

Dabei wird deutlich, dass für das Erstellen des BRIN-Indexes nur ein Bruchteil der Zeit für das Erstellen des B-Tree-Indexes benötigt wurde. Der Ausführungsplan in Listing 11.15 zeigt einen „Bitmap Index Scan“, ebenfalls mit einem Parallelitätsgrad von 5. Kosten und Ausführungszeit sind, verglichen mit dem B-Tree-Index, geringer.

Listing 11.15 Ausführungsplan mit Block Range-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT AVG(t_celsius) FROM temperature
> WHERE t_time > '2020-02-28' AND t_time < '2020-04-01';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=212662.96..212662.97 rows=1 width=32) (actual
time=202.145..233.308 rows=1 loops=1)
   ->  Gather  (cost=212662.44..212662.95 rows=5 width=32) (actual
time=201.919..233.270 rows=6 loops=1)
         Workers Planned: 5
         Workers Launched: 5
         ->  Partial Aggregate  (cost=211662.44..211662.45 rows=1 width=32) (actual
time=154.994..154.995 rows=1 loops=6)
               ->  Parallel Bitmap Heap Scan on temperature  (cost=751.14..210235.30
rows=570853 width=4) (actual time=0.367..109.874 rows=475200 loops=6)
                     Recheck Cond: ((t_time > '2020-02-28 00:00:00'::timestamp
without time zone) AND (t_time < '2020-04-01 00:00:00'::timestamp without time zone))
                     Rows Removed by Index Recheck: 3755
                     Heap Blocks: lossy=3278
                     ->  Bitmap Index Scan on i_temp_brin  (cost=0.00..37.58
rows=2872387 width=0) (actual time=0.824..0.824 rows=183040 loops=1)
                           Index Cond: ((t_time > '2020-02-28 00:00:00'::timestamp
without time zone) AND (t_time < '2020-04-01 00:00:00'::timestamp without time zone))
```

```
 Planning Time: 0.476 ms
 Execution Time: 233.442 ms
(postgres@localhost:5432)[hanser]> SELECT AVG(t_celsius) FROM temperature
> WHERE t_time > '2020-02-28' AND t_time < '2020-04-01';
         avg
---------------------
 50.0099456404130333
(1 row)
Time: 195,167 ms
```

Die Ausführungszeit der SQL-Anweisung wurde etwas mehr als halbiert. Ein besserer Wert ist in diesem Beispiel nicht zu erwarten, da die Zeit für die Aggregation einen Großteil der Gesamtausführungszeit ausmacht. Die Laufzeiten für die Indexscans unterscheiden sich deutlich:

- B-Tree-Indexscan: `Parallel Index Scan using i_temp_btree on temperature (cost=0.56..83743.74 rows=570853 width=4) (`**`actual time=0.248..188.736`**` rows=475200 loops=6)`
- BRIN-Scan: `Bitmap Index Scan on i_temp_brin (cost=0.00..37.58 rows=2872387 width=0) (`**`actual time=0.824..0.824`**` rows=183040 loops=1)`

Das Beispiel ist natürlich ein Idealfall für den BRIN-Index. Die Daten wurden frisch geladen, und zwar in Sortierreihenfolge des Timestamp. Die Verteilung ist damit fast optimal. Die Größe der beiden Indexe unterscheiden sich deutlich.

Listing 11.16 Größenvergleich B-Tree- und BRIN-Index

```
(postgres@localhost:5432)[hanser]> SELECT
>   t.schemaname,
>   t.tablename,
>   c.reltuples::bigint                            AS num_rows,
>   psai.indexrelname                              AS index_name,
>   pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
>   CASE WHEN i.indisunique THEN 'Y' ELSE 'N' END  AS "unique"
> FROM
>   pg_tables t
>   LEFT JOIN pg_class c ON t.tablename = c.relname
>   LEFT JOIN pg_index i ON c.oid = i.indrelid
>   LEFT JOIN pg_stat_all_indexes psai ON i.indexrelid = psai.indexrelid
> WHERE
>   t.schemaname NOT IN ('pg_catalog', 'information_schema')
> ORDER BY 1, 2;
 schemaname |    tablename     | num_rows |      index_name       | index_size |
unique
------------+------------------+----------+-----------------------+------------+-----
---
 public     | temperature      | 31536000 | i_temp_brin           | 72 kB      | N
 public     | temperature      | 31536000 | i_temp_btree          | 676 MB     | N
```

Für die Indexpflege ist Folgendes zu beachten: Wird der Index mit der Option *autosummarize=true* angelegt, erfolgt die Indexpflege automatisch und für den Prozess der Summenbildung kann der AUTOVACUUM-Prozess eingebunden werden. In Folge einer DML-Operation können die folgenden zwei Situationen auftreten:

1. Für die Pages in der Tabelle ist die Summenbildung bereits abgeschlossen. In diesem Fall werden die Summen direkt aktualisiert.

2. Es werden neue Pages erstellt, die noch nicht in die Summenbildung eingeflossen sind. Die Aufgabe wird an den AUTOVACUUM-Prozess übergeben.

TIPP: Aufgrund des höheren Aufwands für die Indexpflege, verglichen mit einem B-Tree-Index, ist er für Tabellen, die häufigen Änderungen auf der indizierten Spalte unterliegen, weniger geeignet. Sein Hauptanwendungsgebiet ist im Data-Warehouse-Umfeld mit großen Tabellen und wenig Änderungen zu sehen.

Der Index ist für viele Datentypen einsetzbar und sowohl für Gleichheitsoperationen als auch Größer- und Kleiner-Vergleiche geeignet.

Schauen wir uns in Listing 11.17 noch den Inhalt einer Page des BRIN-Index mit Hilfe von *pageinspect* an.

Listing 11.17 Inhalt eines BRIN-Index

```
(postgres@localhost:5432)[hanser]> SELECT * FROM
brin_page_items(get_raw_page('i_temp_brin', 5), 'i_temp_brin')
> ORDER BY blknum, attnum LIMIT 10;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |
value
------------+--------+--------+----------+----------+-------------+------------------
----------------------------
          1 | 111744 |      1 | f        | f        | f           | {2020-07-22
01:16:48 .. 2020-07-22 06:51:43}
          2 | 111872 |      1 | f        | f        | f           | {2020-07-22
06:51:44 .. 2020-07-22 12:26:39}
          3 | 112000 |      1 | f        | f        | f           | {2020-07-22
12:26:40 .. 2020-07-22 18:01:35}
          4 | 112128 |      1 | f        | f        | f           | {2020-07-22
18:01:36 .. 2020-07-22 23:36:31}
          5 | 112256 |      1 | f        | f        | f           | {2020-07-22
23:36:32 .. 2020-07-23 05:11:27}
. . .
```

11.3 Hash-Index

Für einen Hash-Index wird aus dem Spaltenwert mit Hilfe einer Hash-Funktion ein 32-Bit-Code erstellt. Ein Hash-Index ist universell einsetzbar und man muss sich wenig Gedanken um den Inhalt der Spalte machen. Es liegt in der Natur der Sache, dass ein Hash-Index nur für Gleichheitsoperationen geeignet ist. Der kostenbasierende Optimizer zieht eine Verwendung in Betracht, wenn die indizierte Spalte für eine Gleichheitsoperation benötigt wird.

Der Index besteht aus Buckets, die wiederum die Hash-Codes und die verbundenen TIDs enthalten. Der Zugriff auf die Buckets wird über eine Bitmap gesteuert. Der Overflow-Bereich wird verwendet, wenn zum Beispiel ein Bucket voll ist und keine neuen Hash-Codes aufnehmen kann. Der Zugriff auf die Buckets wird über eine Bitmap gesteuert. Das bedeutet

für Abfragen, dass alle Overflow Pages gescannt werden müssen. Der VACCUM-Prozess sorgt dafür, dass die Anzahl der Overflow Pages möglichst klein wird. Ist eine Overflow Page leer geworden, wird sie wiederverwendet oder als normale Page eingesetzt. Aktuell gibt es keine Option, einen Hash-Index zu schrumpfen. Das ist nur über einen Neuaufbau zum Beispiel mit dem REINDEX-Befehl möglich.

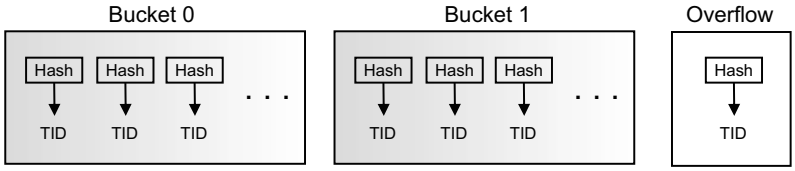


Bild 11.3 Die Architektur des Hash-Index

Ein Hash-Index ist für Tabellen mit häufigen SELECT- und UPDATE-Anweisungen sowie für große Tabellen sehr gut geeignet. Die Zugriffszeiten auf den Index sind sogar geringer als bei einem B-Tree-Index, da der Zugriff direkt auf die Bucket Page erfolgt. Das macht sich insbesondere bei großen Tabellen bemerkbar. Er benötigt weniger I/O-Operationen (logisch und physisch). Aufgrund seiner Einschränkungen steht er nicht in direkter Konkurrenz zum B-Tree-Index. Er kann jedoch für gewisse Fälle besser geeignet sein.

Für Hash-Indexe gelten die folgenden Einschränkungen:

- Eine Verwendung für Unique Constraints ist nicht möglich.
- Sie können nicht als Mehrspaltenindex genutzt werden.
- Die Optionen ASC oder DESC können nicht verwendet werden.
- An geclusterten Tabellen kann kein Hash-Index gebildet werden.

Für das folgende Beispiel ist ein Hash-Index besser geeignet als ein B-Tree-Index. Zunächst legen wir eine Tabelle mit einer Textspalte an.

Listing 11.18 Eine Tabelle mit Textspalte anlegen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE text_big(
>   id      INTEGER,
>   cont    TEXT
> );
CREATE TABLE
```

Diese soll nun mit zufällig generiertem Text gefüllt werden. Zum Generieren des Textes verwenden wir eine PL/pgSQL-Funktion (siehe Listing 11.19).

Listing 11.19 Eine PL/pgSQL-Funktion zum Generieren zufälliger Texte

```
(postgres@localhost:5432)[hanser]> CREATE OR REPLACE FUNCTION str_random(len INTEGER)
> RETURNS text
> AS
> $$
> DECLARE
>   zeichen text[] := '{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,
U,V,W,X,Y,Z,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,-,#,$,?,&,_}';

>   result text := 0;
> BEGIN
```

```
>   FOR i IN 1..len LOOP
>     result := result || zeichen[1+random()*(array_length(zeichen, 1)-1)];
>   END LOOP;
>   RETURN result;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Für das Erzeugen der Testdaten können wir, wie so häufig, die Funktion *generate_series* verwenden. Für den Testfall sind 5 Millionen Sätze ausreichend, um hinreichend gute Performance-Werte zu erhalten. Auf der Tabelle werden ein B-Tree- und ein Hash-Index angelegt.

Listing 11.20 Testdaten mit *generate_series* erzeugen

```
(postgres@localhost:5432)[hanser]> INSERT INTO text_big
> SELECT n, str_random(200)
> FROM generate_series(1, 5000000) x(n);
INSERT 0 5000000
(postgres@localhost:5432)[postgres]>  CREATE INDEX i_text_btree ON text_big (cont);
CREATE INDEX
Time: 175954,122 ms (02:55,954)
(postgresql@localhost:5432)[postgres]> CREATE INDEX i_text_hash ON text_big USING
HASH (cont);
CREATE INDEX
Time: 19891,111 ms (00:19,891)
```

Während das Erstellen des B-Tree-Index fast drei Minuten gedauert hat, konnte der Hash-Index in ca. 20 Sekunden angelegt werden. Der B-Tree-Index muss für jeden Spaltenwert einen Indexwert erstellen und schreiben. Das bewirkt signifikant mehr I/O-Durchsatz bei der Indexerzeugung und schlägt sich auch in der Größe der Indexe nieder (siehe Listing 11.21).

Listing 11.21 Größenunterschied zwischen B-Tree- und Hash-Index

```
(postgres@localhost:5432)[hanser]> SELECT
>   t.schemaname,
>   t.tablename,
>   c.reltuples::bigint                            AS num_rows,
>   psai.indexrelname                              AS index_name,
>   pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
>   CASE WHEN i.indisunique THEN 'Y' ELSE 'N' END  AS "unique"
> FROM
>   pg_tables t
>   LEFT JOIN pg_class c ON t.tablename = c.relname
>   LEFT JOIN pg_index i ON c.oid = i.indrelid
>   LEFT JOIN pg_stat_all_indexes psai ON i.indexrelid = psai.indexrelid
> WHERE
>   t.schemaname NOT IN ('pg_catalog', 'information_schema')
> ORDER BY 1, 2;
 schemaname | tablename | num_rows |  index_name  | index_size | unique
------------+-----------+----------+--------------+------------+--------
 Public     | text_big  |  5000000 | i_text_btree | 1231 MB    | N
 Public     | text_big  |  5000000 | i_text_hash  | 128 MB     | N
```

Für welchen Index wird sich der Query Optimizer entscheiden? Erzeugen Sie den Ausführungsplan mit dem ANALYZE-Befehl.

Listing 11.22 Ausführungsplan mit Hash-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN (ANALYZE, TIMING, VERBOSE, BUFFERS)
> SELECT * FROM text_big WHERE cont = 'Ood9dXtZeZ5X-oBbiK&fb& . . . ';
                                          QUERY PLAN
-----------------------------------------------------------------------------------
-
 Index Scan using i_text_hash on us_sdvz_ad.text_big  (cost=0.00..8.02 rows=1
width=209) (actual time=0.027..0.027 rows=1 loops=1)
   Output: id, cont
   Index Cond: (text_big.cont = 'Ood9dXtZeZ5X-oBbiK&fb& . . . ¸::text)
```

Die Entscheidung ist klar für die geringeren Kosten und damit für den Hash-Index gefallen. Eine Abfrage mit 1000 Sätzen (Listing 11.23) ergibt eine Ausführungszeit von ca. 6 Millisekunden mit dem Hash-Index. Der B-Tree-Index benötigt ungefähr die zehnfache Zeit. Es sind mehr I/O-Operationen erforderlich und die Vergleiche eines langen Strings sind aufwendiger als die des Hash-Codes.

Listing 11.23 Ausführungszeit mit Hash- und B-Tree-Index

```
(postgres@localhost:5432)[hanser]> SELECT * FROM text_big WHERE cont IN
> (SELECT cont FROM text_big LIMIT 1000);
. . .
(1000 rows)
Time: 5,761 ms
(postgres@localhost:5432)[postgres]> DROP INDEX i_text_hash;
(postgres@localhost:5432)[postgres]> SELECT * FROM text_big WHERE cont IN
> (SELECT cont FROM text_big LIMIT 1000);
. . .

(1000 rows)
Time: 56,231 ms
```

Auch wenn das Beispiel so gewählt wurde, dass der Hash-Index seine Vorzüge ausspielen kann, macht es deutlich, dass es durchaus Anwendungsfälle für den Hash-Index gibt.

Der Inhalt eines Bucket ist relativ einfach und kann mit der Funktion *hash_page_items* angezeigt werden.

Listing 11.24 Den Inhalt eines Buckets eines Hash-Index anzeigen

```
(postgres@localhost:5432)[hanser]> SELECT * FROM hash_page_items(get_raw_page('big_
hash_index', 1)) LIMIT 10;
 itemoffset |    ctid     |   data
------------+-------------+----------
          1 | (185482,50) | 15269888
          2 | (160014,16) | 15269888
          3 | (78288,43)  | 15269888
          4 | (39763,51)  | 15269888
          5 | (220334,82) | 18153472
          6 | (200499,14) | 18153472
          7 | (122727,16) | 18153472
          8 | (80516,60)  | 18153472
          9 | (208958,26) | 55312384
         10 | (183200,49) | 55312384
```

11.4 Generalized Inverted Index (GIN)

Dieser Indextyp ist besonders für Daten geeignet, die aus zusammengesetzten Werten bestehen. Er kommt häufig für die Textverarbeitung zum Einsatz. Ein Text kann als Zusammensetzung von Worten und Begriffen verstanden werden. Soll nach bestimmten Worten innerhalb der Texte gesucht werden, dann ist ein BRIN-Index sehr gut geeignet.

HINWEIS: Auch wenn ein GIN-Index in der Textverarbeitung häufig verwendet wird, soll darauf hingewiesen werden, dass sich sein Einsatzgebiet nicht darauf beschränkt. Er ist für sehr viele Datentypen geeignet, unter anderem für Zahlen, Arrays, JSON- oder auch Zeitformate oder Binärdaten. Zusätzlich ist er so offen gestaltet, dass es möglich ist, ihn für eigene Datentypen zu verwenden.

Ein GIN-Index beinhaltet jeweils eine Relation von Schlüsselbegriff und einer Liste von TIDs (Posting-Liste), in denen dieser Begriff vorkommt. Damit ist klar, dass eine TID in mehreren Posting-Listen vorkommen kann. Allerdings wird jeder Schlüsselwert im Index nur einmal gespeichert. Das macht den Index sehr kompakt und beschränkt das Wachstum.

Für das folgende Beispiel erstellen wir zunächst eine Tabelle mit zufällig generiertem Text. Die Erzeugung des Textes erfolgt aus einer Datei, die Wörter enthält. Dazu verwenden wir ein Python-Programm, das einfach zu erstellen ist. Eine Textdatei mit Wörtern kann aus dem Internet heruntergeladen werden.

Listing 11.25 Eine Testtabelle mit Textspalte erstellen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE dokument(id INTEGER, inhalt text);
CREATE TABLE
```

Listing 11.26 Python-Programm zum Laden von Texten aus zufälligen Wörtern

```
import random
import psycopg2
def random_line(fname):
    lines = open(fname).read().splitlines()
    return random.choice(lines)
def line():
    str = ""
    for i in range(1, 10):
        str = str + " " + random_line('words.txt')
    return str
conn = psycopg2.connect(host="localhost", database="hanser", user="postgres",
password="postgres")
curs = conn.cursor()
for j in range(1, 1000):
    curs.execute("INSERT INTO dokument(id, inhalt) VALUES (%s,%s)", (j, line()))
conn.commit()
```

Für die Suche nach allen Sätzen, die ein bestimmtes Wort enthalten, führt PostgreSQL einen Full Scan (Sequential Scan) der Tabelle aus. Der Ausführungsplan in Listing 11.27 basiert auf 2,5 Millionen Sätzen.

Listing 11.27 Ausführungsplan ohne GIN-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM dokument
> WHERE to_tsvector('english', inhalt) @@ to_tsquery('english', 'befunden');
                                          QUERY PLAN
-------------------------------------------------------------------------------------
--
 Gather  (cost=1000.00..211416.17 rows=12799 width=98) (actual time=8.160..15723.192
rows=1 loops=1)
   Workers Planned: 4
   Workers Launched: 4
   ->  Parallel Seq Scan on dokument  (cost=0.00..209136.27 rows=3200 width=98)
(actual time=12492.955..15628.792 rows=0 loops=5)
         Filter: (to_tsvector('english'::regconfig, inhalt) @@
'''befunden'''::tsquery)
         Rows Removed by Filter: 511949
```

PostgreSQL liest also die gesamte Tabelle sequenziell und sucht die relevanten Sätze über einen Filter heraus. Es werden vier Worker-Prozesse verwendet, die parallel lesen und filtern. Das erzeugt nicht nur viele I/O-Operationen, sondern auch einen großen CPU-Verbrauch für den Filter. Die Ausführungszeit beträgt 16 Sekunden.

Listing 11.28 Textabfrage mit Sequential Scan

```
(postgres@localhost:5432)[hanser]> SELECT * FROM dokument
> WHERE to_tsvector('english', inhalt) @@ to_tsquery('english', 'befunden');
   id    |                      inhalt
---------+-------------------------------------------------
 9999999 | Das Dokument wurde geprueft und fuer gut befunden
(1 row)
Time: 16014,359 ms (00:16,014)
```

Der CPU-Verbrauch lag über die gesamte Zeit bei 80 % Auslastung. Falls solche Abfragen von mehreren Sessions gleichzeitig ausgeführt werden, geht das System schnell wegen Überlastung in die Knie. Auch ein normaler B-Tree-Index und eine Bedingung mit LIKE würden die Ausführung nicht beschleunigen und deshalb vom Query Optimizer nicht gewählt werden.

Wie bereits erwähnt, arbeitet das Indexsystem von PostgreSQL mit Operatorenklassen. Wenn man vor der Wahl steht, herauszufinden, welcher Indextyp geeignet ist, sollte man sich zuerst an Operatorenklassen orientieren. Für unser Beispiel erfolgt das mit einer Abfrage der verfügbaren Indextypen für die Klasse „@@ tsquery“ (Listing 11.29).

Listing 11.29 Indextypen für Operatorenklasse abfragen

```
(postgres@localhost:5432)[hanser]> SELECT am.amname, f.opfname,
> op.amopopr::regoperator
> FROM pg_am am, pg_opfamily f,pg_amop op
> WHERE f.opfmethod = am.oid AND op.amopfamily = f.oid
> AND op.amopopr = '@@(tsvector,tsquery)'::regoperator;
 amname |   opfname    |       amopopr
--------+--------------+---------------------
 gist   | tsvector_ops | @@(tsvector,tsquery)
 gin    | tsvector_ops | @@(tsvector,tsquery)
```

Erstellen wir nun einen GIN-Index über die Textspalte. Mit der Funktion *to_tsvector* wird der Text in die einzelnen Worte zerlegt. Das ist ein Format, dass der GIN-Index verarbeiten kann.

Listing 11.30 Einen GIN-Index für eine Textspalte erstellen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_dokument_gin
ON dokument USING GIN (to_tsvector('english', inhalt));
CREATE INDEX
```

Der Ausführungsplan in Listing 11.31 zeigt deutlich geringere Kosten und auf eine parallele Ausführung wird verzichtet.

Listing 11.31 Ausführungsplan mit GIN-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM dokument
> WHERE to_tsvector('english', inhalt) @@ to_tsquery('english', 'befunden');
                                        QUERY PLAN
-----------------------------------------------------------------------------------------
-
 Bitmap Heap Scan on dokument  (cost=135.19..30562.27 rows=12799 width=98) (actual
time=0.062..0.063 rows=1 loops=1)
   Recheck Cond: (to_tsvector('english'::regconfig, inhalt) @@
'''befunden'''::tsquery)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on i_dokument_gin  (cost=0.00..131.99 rows=12799 width=0)
(actual time=0.058..0.058 rows=1 loops=1)
         Index Cond: (to_tsvector('english'::regconfig, inhalt) @@
'''befunden'''::tsquery)
```

Die Ausführungszeit reduziert sich auf unter eine Millisekunde, da die Daten über den GIN-Index effektiv gefunden werden. Die Belastung der Systemressourcen ist minimal.

Listing 11.32 Textabfrage mit GIN-Index

```
(postgres@localhost:5432)[hanser]> SELECT * FROM dokument
> WHERE to_tsvector('english', inhalt) @@ to_tsquery('english', 'befunden');
   id    |                      inhalt
---------+-------------------------------------------------
 9999999 | Das Dokument wurde geprueft und fuer gut befunden
(1 row)
Time: 0,676 ms
```

TIPP: Solche Abfragen nach Schlüsselwörtern in Texten kommen in der Praxis häufig vor. Mit einem normalen B-Tree-Index lassen sich die einzelnen Wörter nicht darstellen. Sequenzielle Scans haben einen hohen Ressourcenverbrauch und lange Ausführungszeiten. Ein GIN-Index ist auf Grund seiner Struktur auch für große Textfelder noch sehr kompakt.

Ein Nachteil bei der Verwendung von GIN-Indexen ist, dass Updates aufwändig sind und viele Änderungen im Index nach sich ziehen. Wenn komplette Texte ausgetauscht werden, kann man sich den hohen Änderungsaufwand gut vorstellen.

Ein GIN-Index kann mit der Option *fastupdate=true* versehen werden. Ist das Feature eingeschaltet, werden Änderungen zunächst in eine separate Liste geschrieben. Überschreitet die Liste die Größe von dem im Parameter *gin_pending_list_limit* festgelegten Wert, dann werden die aufgelaufenen Änderungen eingepflegt. Das geschieht auch während eines VACUUM-Laufs. Die Option hat den Nachteil, dass während einer Abfrage die separate Liste zusätzlich verarbeitet werden muss. Sie kann jedoch ein guter Kompromiss sein.

Für GIN-Indexe besteht die Einschränkung, dass Full Index Scans nicht unterstützt werden.

11.5 Generalized Search Tree-Index (GiST)

Ein GiST-Index ist ein sogenannter ausbalancierter Index mit Baumstruktur. Er unterscheidet sich vom B-Tree-Index durch umfangreichere Vergleichsmöglichkeiten. Ein B-Tree-Index ist im Wesentlichen auf Operationen wie „gleich“, größer“ oder „kleiner“ beschränkt. Die Anforderungen an ein modernes Datenbanksystem sind jedoch gewachsen. Denken Sie an Textdokumente, Bilder, Sprachaufzeichnungen, Geodaten oder auch selbst definierte Datentypen. Auch für solche Daten werden geeignete Indexe benötigt.

Und hier kommt der GiST-Index ins Spiel. Für ihn können Regeln für die Verteilung von Daten beliebigen Typs über eine ausbalancierte Baumstruktur festgelegt werden. Jeder Leaf-Knoten enthält einen logischen Ausdruck und zeigt auf eine Zeile in der Tabelle (TID). Die Suche in einem GiST-Index verwendet eine Konsistenzfunktion, die über das Interface definiert wird. Die Funktion wird aufgerufen und liefert ein Ergebnis, das auf Konsistenz mit dem Suchprädikat geprüft wird.

Schauen wir uns ein Beispiel für eine Tabelle mit geometrischen Datentypen an. Die Tabelle in Listing 11.33 verwendet die Datentypen BOX und POINT, die ein Rechteck und einen zweidimensionalen Punkt (x- und y-Koordinaten) interpretieren.

Listing 11.33 Eine Tabelle mit geometrischen Datentypen anlegen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE geom(
> rechteck BOX,
> punkt    POINT);
CREATE TABLE
(postgres@localhost:5432)[postgres]> INSERT INTO geom
> SELECT BOX(POINT(0.05*n, 0.05*n), POINT(0.05*n, 0.05*n)),
>   POINT(0.05*n, 0.05*n)
> FROM generate_series(0,10000000) AS n;
INSERT 0 10000001
```

Eine SELECT-Anweisung soll die Sätze liefern, die das Rechteck in einem bestimmten Bereich darstellen. Dazu wird der Operator „<@“ verwendet. Ohne GiST-Index führt der Server einen Full Table Scan durch (siehe Listing 11.34). Die Ausführungszeit beträgt ungefähr eine halbe Sekunde.

Listing 11.34 Abfrage einer geometrischen Spalte ohne Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT rechteck FROM geom
```

```
> WHERE rechteck <@ BOX(POINT(8,8), POINT(9,9));
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..147541.34 rows=10000 width=32) (actual time=0.554..456.739
rows=21 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on geom  (cost=0.00..145541.34 rows=4167 width=32) (actual
time=253.794..396.067 rows=7 loops=3)
         Filter: (rechteck <@ ,(9,9),(8,8)'::box)
         Rows Removed by Filter: 3333327
 Planning Time: 0.105 ms
 Execution Time: 456.760 ms
```

Legen Sie einen GiST-Index auf der Spalte „rechteck“ an (Listing 11.35). PostgreSQL kann diesen Spaltentyp im Index verwenden. Die Definition eines eigenen Operators und einer eigenen Funktion wären an dieser Stelle auch möglich. Um das Beispiel transparent zu halten, verwenden wir einen Standardoperator, den PostgreSQL bereits kennt.

Listing 11.35 Einen GiST-Index anlegen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_gist_geom
> ON geom USING GIST (rechteck);
CREATE INDEX
```

Jetzt führt der Server einen Index Only Scan durch. Mit der B-Tree-Struktur ist es möglich, das Ergebnis schnell einzugrenzen. Die Ausführungszeit beträgt 0,3 Millisekunden.

Listing 11.36 Ausführungsplan mit GiST-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT rechteck FROM geom
> WHERE rechteck <@ BOX(POINT(8,8), POINT(9,9));
                                  QUERY PLAN
-------------------------------------------------------------------------------
-
 Index Only Scan using i_gist_geom on geom  (cost=0.42..663.42 rows=10000 width=32)
(actual time=0.289..0.310 rows=21 loops=1)
   Index Cond: (rechteck <@ '(9,9),(8,8)'::box)
   Heap Fetches: 0
 Planning Time: 0.697 ms
 Execution Time: 0.325 ms
```

Die Größe des Index lässt sich wieder mit der SQL-Abfrage aus Listing 11.21 ermitteln. Sie liegt in etwa im Bereich eines normalen B-Tree-Index.

Listing 11.37 Die Größe des GiST-Index

```
schemaname | tablename | num_rows | index_name  | index_size | unique
-----------+-----------+----------+-------------+------------+--------
 public    | geom      | 10000001 | i_gist_geom | 953 MB     | N
```

Der GiST-Index ist nicht nur bei der Verwendung einer WHERE-Klausel hilfreich, sondern auch für die Sortierung mit ORDER BY. Die Abfrage in Listing 11.38 zeigt die Operatoren, die standardmäßig durch einen GiST-Index für den Spaltentyp BOX unterstützt werden. Erweiterungen durch selbst definierte Operatoren sind möglich.

Listing 11.38 Standardoperatoren für eine BOX-Spalte eines GiST-Index

```
(postgres@localhost:5432)[hanser]> SELECT amop.amopopr::regoperator, amop.
amoppurpose, amop.amopstrategy
> FROM pg_opclass opc, pg_opfamily opf, pg_am am, pg_amop amop
> WHERE opc.opcname = 'box_ops'
> AND opf.oid = opc.opcfamily
> AND am.oid = opf.opfmethod
> AND amop.amopfamily = opc.opcfamily
> AND am.amname = 'gist'
> AND amop.amoplefttype = opc.opcintype;
    amopopr     | amoppurpose | amopstrategy
----------------+-------------+--------------
 <->(box,point) | o           |           15
 <<(box,box)    | s           |            1
 &<(box,box)    | s           |            2
 &&(box,box)    | s           |            3
 &>(box,box)    | s           |            4
 >>(box,box)    | s           |            5
 ~=(box,box)    | s           |            6
 @>(box,box)    | s           |            7
 <@(box,box)    | s           |            8
 &<|(box,box)   | s           |            9
 <<|(box,box)   | s           |           10
 |>>(box,box)   | s           |           11
 |&>(box,box)   | s           |           12
```

Auch für GiST-Indexe ist eine interne Betrachtung mit der Erweiterung *pageinspect* möglich. Listing 11.39 liefert Informationen über die im Index gespeicherten Daten.

Listing 11.39 Daten einer Page im GiST-Index auslesen

```
(postgres@localhost:5432)[hanser]> SELECT ctid, keys FROM gist_page_items(get_raw_
page('i_gist_geom',0), 'i_gist_geom');
      ctid      |                          keys
----------------+-------------------------------------------------------------
 (14030,65535)  | (rechteck)=((28589.3,28589.3),(0,0))
 (14031,65535)  | (rechteck)=((57178.65,57178.65),(28589.35,28589.35))
 (21004,65535)  | (rechteck)=((85768,85768),(57178.7,57178.7))
 (27977,65535)  | (rechteck)=((114357.35,114357.35),(85768.05,85768.05))
 (34950,65535)  | (rechteck)=((142946.7,142946.7),(114357.4,114357.4))
 (41923,65535)  | (rechteck)=((171536.05,171536.05),(142946.75,142946.75))
 (48896,65535)  | (rechteck)=((200125.4,200125.4),(171536.1,171536.1))
. . .
```

Die Rohdaten zeigen den Inhalt einer Page im Hexadezimalformat (siehe Listing 11.40).

Listing 11.40 Auslesen der Rohdaten eines GiST-Index

```
(postgres@localhost:5432)[hanser]> SELECT ctid, key_data
> FROM gist_page_items_bytea(get_raw_page('i_gist_geom', 0));
      ctid      |                                        key_data
----------------+-------------------------------------------------------------------
 (14030,65535)  | \x0000ce36ffff28003333333353ebdb403333333353ebdb400000000000000
 (14031,65535)  | \x0000cf36ffff2800cdcccccc54ebeb40cdcccccc54ebeb406666666656ebd
 (21004,65535)  | \x00000c52ffff28000000000080f0f4400000000080f0f4406666666656ebe
 (27977,65535)  | \x0000496dffff28009a99999955ebfb409a99999955ebfb40cdcccccc80f0f
 (34950,65535)  | \x00008688ffff28009a999999157301419a99999915730141666666656ebf
```

```
 (41923,65535)  | \x0000c3a3ffff2800666666666680f00441666666666680f00441000000000016730
 (48896,65535)  | \x000000bfffff2800333333333eb6d084133333333eb6d0841cdcccccc80f00
. . .
```

11.6 Expression-Index

Ein Expression-Index besitzt die Architektur eines B-Tree-Index. Das Ergebnis des Ausdrucks oder der Funktion wird im Index gespeichert und dem entsprechenden Datensatz zugeordnet. Für SQL-Abfragen ergeben sich signifikante Performancevorteile, da der Ausdruck sonst für jeden Datensatz oder Indexeintrag berechnet werden müsste.

Diesen Vorteil erkauft man sich für einen Mehraufwand bei der Indexerstellung. Auch die Pflege des Index erhöht sich um den Aufwand der Neuberechnung des Ergebnisses. Für das Beispiel in Listing 11.41 wird eine Tabelle mit einer Spalte vom Typ „timestamp“ erstellt.

Listing 11.41 Eine Tabelle für einen Expression-Index erstellen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE t_time AS
> SELECT zeit, 'Some text.............'::text
> FROM generate_series(timestamp '2020-01-01',
> timestamp '2021-01-01', interval '1 second') s(zeit);
SELECT 31622401
```

Eine SQL-Abfrage soll alle Zeitstempel selektieren, die den ersten Tag eines Monats enthalten. Dafür wird die Funktion EXTRACT verwendet. Ein normaler B-Tree-Index wurde für die Abfrage nicht berücksichtigt. Die Abfrage führt einen Full Table Scan durch (siehe Listing 11.42).

Listing 11.42 SQL-Abfrage mit Zeit-Funktion

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM t_time WHERE EXTRACT(day FROM zeit) = 1;
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Gather  (cost=1000.00..344196.40 rows=158112 width=40) (actual time=0.828..2235.723
rows=1036801 loops=1)
   Workers Planned: 5
   Workers Launched: 5
   ->  Parallel Seq Scan on t_time  (cost=0.00..327385.20 rows=31622 width=40)
(actual time=128.198..2072.134 rows=172800 loops=6)
         Filter: (EXTRACT(day FROM zeit) = '1'::numeric)
         Rows Removed by Filter: 5097600
 Planning Time: 1.208 ms
 Execution Time: 2268.116 ms
```

Die Ausführungszeit beträgt mehr als zwei Sekunden. Nach Anlegen eines Expression-Index führt PostgreSQL einen Indexscan mit einem Bitmap-Heap-Scan durch. Die Ausführungszeit reduziert sich auf ungefähr 200 Millisekunden.

Listing 11.43 Ausführungsplan mit Expression-Index

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_expr_t_time
ON t_time ((EXTRACT(day FROM zeit)));
CREATE INDEX
(postgresql@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM t_time WHERE EXTRACT(day FROM zeit) = 1;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_time  (cost=2965.93..225150.54 rows=158112 width=31) (actual
time=96.770..202.794 rows=1036801 loops=1)
   Recheck Cond: (EXTRACT(day FROM zeit) = '1'::numeric)
   Heap Blocks: exact=7635
   ->  Bitmap Index Scan on i_expr_t_time  (cost=0.00..2926.40 rows=158112 width=0)
(actual time=95.794..95.794 rows=1036801 loops=1)
         Index Cond: (EXTRACT(day FROM zeit) = '1'::numeric)
 Planning Time: 1.500 ms
 Execution Time: 223.338 ms
```

TIPP: Die Verwendung von Expression-Indexen hängt stark davon ab, welche SQL-Abfragen verwendet werden. Prüfen Sie regelmäßig die Verwendung der Indexe, so wie in Listing 11.1 beschrieben. Die Indexpflege ist aufwendiger als für einen normalen B-Tree-Index.

11.7 Partieller Index

Ein partieller Index wird über einen Teil der Tabelle gebildet. Die Teilmenge der Tabelle wird mit Hilfe einer Bedingung gebildet, die auch Prädikat des Index genannt wird. Der Index besteht also nur aus Einträgen, die dem Prädikat genügen.

Mit partiellen Indexen kann die Größe des Index signifikant reduziert und die Laufzeit von SQL-Abfragen verbessert werden. Auch der Aufwand für die Indexpflege in Zusammenhang mit DML-Anweisungen ist kleiner.

Der häufigste Grund für die Verwendung eines partiellen Index ist, die Indizierung von allgemeinen Inhalten, also von Inhalten, die einen großen Teil der Tabelle ausmachen, zu vermeiden. SQL-Abfragen nach allgemeinen Inhalten lösen ohnehin einen Full Table Scan aus und verwenden den Index nicht. Es macht also keinen Sinn, diese Sätze im Index zu pflegen.

Ein typischer Anwendungsfall ist, das Indizieren von Nullwerten zu vermeiden, wenn sie einen Großteil der Tabelle ausmachen. Für das Beispiel wird eine Tabelle erzeugt, in der jeder zehnte Eintrag das Löschkennzeichen „N“ besitzt. Für Abfragen sind nur Sätze mit Löschkennzeichen „N“ von Interesse. Dafür bietet sich ein partieller Index an. Zusammen mit der Tabelle werden ein normaler B-Tree-Index und ein partieller Index erstellt.

Listing 11.44 Tabelle für partiellen Index erstellen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE text_part AS
> SELECT n AS id,
> CASE n % 10
>   WHEN 0 THEN 'N'
>   ELSE 'Y'
> END AS geloescht,
> str_random(200) AS cont
> FROM generate_series(1, 5000000) x(n);
SELECT 5000000
(postgres@localhost:5432)[hanser]> CREATE INDEX i_btree_text ON text_part(cont);
CREATE INDEX
(postgres@localhost:5432)[hanser]> CREATE INDEX i_part_text ON text_part(cont)
> WHERE geloescht = ,N';
CREATE INDEX
```

Ein Größenvergleich mit Hilfe der Abfrage aus Listing 11.16 zeigt einen deutlichen Unterschied.

Listing 11.45 Größenvergleich mit partiellem Index

```
 schemaname |  tablename  | num_rows |   index_name   | index_size | unique
------------+-------------+----------+----------------+------------+--------
 public     | text_part   |  5000028 | i_btree_text   | 1231 MB    | N
 public     | text_part   |  5000028 | i_part_text    | 123 MB     | N
```

Die Ausführungszeiten unterscheiden sich nur unwesentlich für die beiden Indextypen. Der Größenunterschied ist jedoch erheblich.

Seit der Version 11 können B-Tree-Indexe mit einer INCLUDE-Klausel gebildet werden. Die in der Klausel genannten Spalten werden zusätzlich in den Index aufgenommen, sind aber nicht Bestandteil des Schlüssels. Der Index wird damit zwar größer. Der Vorteil entsteht für SQL-Abfragen, die sich auf diese Spalten beziehen. Anstelle des üblichen Algorithmus, der das Lesen des Indexschlüssels und das anschließende Lesen des zugehörigen Satzes aus der Tabelle vorsieht, kann ein Index-only-Scan ausgeführt werden. Dieser bietet signifikante Performance-Vorteile. Das Lesen aus Index und Tabelle ist mit vielen Single-Block Read-Lesezugriffen verbunden und als Performance-Killer bekannt.

In Listing 11.46 erfolgt eine SQL-Abfrage der Tabelle *sales* mit einem B-Tree-Index. Da die Spalte „amount“ nicht Bestandteil des Index ist, erfolgt zusätzlich ein Zugriff auf die Tabelle mit einem Bitmap-Heap-Scan. Die Ausführungszeit beträgt ungefähr fünf Sekunden.

Listing 11.46 SQL-Abfrage mit B-Tree-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT sum(amount) FROM sales WHERE customer_id = 99;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=523427.44..523427.45 rows=1 width=32) (actual
time=5012.670..5043.815 rows=1 loops=1)
   ->  Gather  (cost=523426.79..523427.40 rows=6 width=32) (actual
time=5011.108..5043.789 rows=7 loops=1)
         Workers Planned: 6
         Workers Launched: 6
```

```
         ->  Partial Aggregate  (cost=522426.79..522426.80 rows=1 width=32) (actual
time=4931.776..4931.776 rows=1 loops=7)
               ->  Parallel Bitmap Heap Scan on sales  (cost=6089.23..522199.01
rows=91111 width=6) (actual time=113.528..4860.370 rows=71429 loops=7)
                     Recheck Cond: (customer_id = 99)
                     Heap Blocks: exact=45905
                     ->  Bitmap Index Scan on i_cust_sales  (cost=0.00..5952.57
rows=546667 width=0) (actual time=119.542..119.543 rows=500000 loops=1)
                           Index Cond: (customer_id = 99)
 Planning Time: 1.167 ms
 Execution Time: 5043.932 ms
```

Nach Anlegen eines Index mit Aufnahme der Spalte „amount“ als Nicht-Schlüssel wird die Anweisung erneut ausgeführt. PostgreSQL führt jetzt einen Index-Only-Scan aus mit einer Ausführungszeit von 144 Millisekunden.

Listing 11.47 Index mit INCLUDE-Klausel

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_cust_sales_incl ON
> sales(customer_id) INCLUDE(amount);
CREATE INDEX
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT sum(amount) FROM sales WHERE customer_id = 99;
                                          QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=15229.34..15229.35 rows=1 width=32) (actual
time=117.679..144.446 rows=1 loops=1)
   ->  Gather  (cost=15228.90..15229.31 rows=4 width=32) (actual
time=117.562..144.435 rows=5 loops=1)
         Workers Planned: 4
         Workers Launched: 4
         ->  Partial Aggregate  (cost=14228.90..14228.91 rows=1 width=32) (actual
time=77.703..77.704 rows=1 loops=5)
               ->  Parallel Index Only Scan using i_cust_sales_incl on sales
(cost=0.56..13887.23 rows=136667 width=6) (actual time=1.151..67.076 rows=100000
loops=5)
                     Index Cond: (customer_id = 99)
                     Heap Fetches: 0
 Planning Time: 0.932 ms
 Execution Time: 144.484 ms
```

11.8 Individueller Index

PostgreSQL ist als ein sehr offenes System mit vielen Erweiterungsmöglichkeiten bekannt. Das Prinzip macht auch bei den Indexen nicht halt. Unter Beachtung bestimmter Rahmenbedingungen ist es möglich, einen eigenen Indextyp zu erstellen und in den Server einzubinden. Keines der anderen populären Datenbankbetriebssysteme bietet diese Möglichkeit. Das folgende Beispiel zeigt eindrucksvoll, wie ein solcher Index entwickelt und eingesetzt werden kann. Es handelt sich um einen Index mit Verwendung eines Bloom-Filters. Dieser Indextyp ist das Zusatzmodul „bloom“ und kann als Erweiterung einfach installiert werden.

TIPP: Wenn Sie planen, einen eigenen, individuellen Indextyp zu erstellen, dann ist die Erweiterung „bloom“ eine gute Grundlage und zeigt sehr gut auf, wie vorzugehen ist und welche Funktionen geschrieben werden müssen. ■

Bloom-Filter sind im Datenbankumfeld sehr beliebt und können vielseitig eingesetzt werden. Mit ihrer Hilfe kann sehr effektiv festgestellt werden, ob sich Daten einem Wert zuordnen lassen. Bloom-Filter verbrauchen im Vergleich zu anderen Methoden sehr wenig Speicherplatz. Zu beachten ist nur, dass falsch positive Werte möglich sind. Diese können jedoch am Ende des Prozesses einfach herausgefiltert werden.

Ein Bloom-Filter besteht aus einem n-stelligen Bit-Array (Bitmap), das am Anfang mit Nullen gefüllt ist. Mit Hilfe von mehreren Hashfunktionen, die jeweils ein Ergebnis im Wertebereich von null bis „n-1“ haben, wird die entsprechende Position im Bitmap auf „1“ gesetzt. Soll überprüft werden, ob ein bestimmter Wert enthalten ist, wird ebenfalls das Ergebnis der Hashfunktionen ermittelt und mit dem Bitmap verglichen. Enthält eine der Positionen im Bitmap eine Null, dann kann ausgeschlossen werden, dass dieser Wert enthalten ist.

Beim Erstellen des Index können zwei Parameter angegeben werden:

- Die Länge der Signatur in Bits. Die Signatur befindet sich in jedem einzelnen Indexeintrag und stellt einen Indexeintrag dar. Der Standardwert ist 80 und der Maximalwert ist 4096.
- Anzahl von Bits für jede einzelne Indexspalte mit einem Standardwert von 2 und einem Maximum von 4095.

Die Größe der Signatur und die Anzahl von Bits pro Spalte beeinflussen die Wahrscheinlichkeit von falsch positiven Ergebnissen. Größere Werte verringern die Wahrscheinlichkeit, vergrößern aber den Index und die Zeit für Abfragen. Für besondere Fälle kann es Sinn machen, mit den Werten zu experimentieren. Es existieren auch mathematische Formeln zur Berechnung der Wahrscheinlichkeit. In vielen Fällen sind die Standardwerte ein guter Kompromiss.

Mit dem Einfügen eines neuen Satzes in die Tabelle wird eine Signatur erstellt und der Index aktualisiert. Ein Standard-Bloom-Filter unterstützt nicht das Löschen von Elementen. Das ist jedoch für den Index unerheblich. Wird ein Satz in der Tabelle gelöscht, dann wird die Signatur im Index gelöscht.

Wie beim Hash-Index liegt es in seiner Natur, dass ein Bloom-Filter-Index nur Gleichheitsoperationen unterstützt. Für SQL-Abfragen ist er sehr effektiv. Er besitzt eine flache Struktur und ist klein. Der Scan-Prozess findet fast ausschließlich im Memory statt.

Das folgende Beispiel ist gewählt, um die Vorzüge eines Bloom-Filter-Index hervorzuheben. Der Index ist nicht Bestandteil der Standarddistribution. Der erste Schritt ist, die Erweiterung zu installieren.

Listing 11.48 Die Erweiterung „Bloom-Filter-Index“ installieren

```
(postgres@localhost:5432)[hanser]> CREATE EXTENSION bloom;
CREATE EXTENSION
```

Die Tabelle für den Test besteht aus mehreren Spalten vom Typ *integer*. Solche Tabellen kommen in der Praxis häufiger vor. SQL-Abfragen bestehen aus Kombinationen der Spalten in der WHERE-Bedingung. Für das Laden der Testdaten verwenden wir wieder die Funktion

generate_series. Im Beispiel werden 50 Millionen Sätze (ca. 7 GByte) geladen. Das Beispiel lässt sich auch mit weniger Sätzen gut nachvollziehen.

Listing 11.49 Laden einer Tabelle für den Bloom-Filter-Index

```
(postgres@localhost:5432)[hanser]> CREATE TABLE tbloom AS
> SELECT
> (random() *4000000)::int as i1,
> (random() *4000000)::int as i2,
> (random() *4000000)::int as i3,
> (random() *4000000)::int as i4,
> (random() *4000000)::int as i5,
> (random() *4000000)::int as i6,
> (random() *4000000)::int as i7,
> (random() *4000000)::int as i8,
> (random() *4000000)::int as i9,
> (random() *4000000)::int as i10,
> (random() *4000000)::int as i11,
> (random() *4000000)::int as i12,
> (random() *4000000)::int as i13,
> (random() *4000000)::int as i14,
> (random() *4000000)::int as i15,
> (random() *4000000)::int as i16,
> 'Text.....................................'
> FROM
> generate_series(1,50000000);
SELECT 50000000
```

Zum Vergleich wird zuerst ein B-Tree-Index erstellt. Für ein optimales Ergebnis müsste für jede Spalte ein Index erstellt werden. Wir verzichten hier jedoch darauf und erstellen einen kombinierten Index für alle Spalten, mit dem Nachteil, dass der Index vom Optimizer nicht benutzt wird. Der Index besitzt eine beachtliche Größe von ungefähr 4 GByte, was mehr als 50 Prozent der Größe der Tabelle entspricht.

Listing 11.50 Einen B-Tree-Index auf der Bloom-Filter-Tabelle anlegen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX btreeidx
> ON tbloom (i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16);
CREATE INDEX
(postgres@localhost:5432)[hanser]> SELECT pg_size_pretty(pg_relation_
size('btreeidx'));
 pg_size_pretty
----------------
 4083 MB
```

Die SQL-Abfrage in Listing 11.51 führt einen Full Table Scan durch, ohne den B-Tree-Index zu benutzen. Die Ausführungszeit beträgt ungefähr vier Sekunden.

Listing 11.51 Ausführungsplan ohne Bloom-Filter-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM tbloom WHERE i2 = 31286 AND i5 = 209009
> AND i7 = 653944 AND i9 = 256927 AND i10 = 285508 AND i12 = 93431;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Gather  (cost=1000.00..1430924.43 rows=1 width=115) (actual time=3841.300..3871.918
rows=0 loops=1)
   Workers Planned: 2
```

```
   Workers Launched: 2
   ->  Parallel Seq Scan on tbloom  (cost=0.00..1429924.33 rows=1 width=115) (actual
time=3814.317..3814.317 rows=0 loops=3)
         Filter: ((i2 = 31286) AND (i5 = 209009) AND (i7 = 653944) AND (i9 = 256927)
AND (i10 = 285508) AND (i12 = 93431))
         Rows Removed by Filter: 16666667
 Planning Time: 0.112 ms
 Execution Time: 3871.948 ms
```

Erstellen wir nun einen Bloom-Filter-Index. Es fällt auf, dass der Index wesentlich kleiner als der B-Tree-Index ist.

Listing 11.52 Einen Bloom-Filter-Index erstellen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX bloomidx ON tbloom
> USING bloom (i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15,
i16);
CREATE INDEX
(postgres@localhost:5432)[hanser]> SELECT
pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 766 MB
```

Der Ausführungsplan in Listing 11.53 zeigt, dass der Optimizer den Bloom-Filter-Index verwendet und einen Bitmap-Heap-Scan auf der Tabelle durchführt. Dieser Prozess ist mit wesentlich geringeren Kosten und Ressourcenverbrauch verbunden. Auch die Ausführungszeit wird auf ungefähr 680 Millisekunden reduziert.

Listing 11.53 Ausführungsplan mit Bloom-Filter-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT * FROM tbloom WHERE i2 = 31286 AND i5 = 209009
> AND i7 = 653944 AND i9 = 256927 AND i10 = 285508 AND i12 = 93431;
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=1392164.00..1392168.03 rows=1 width=115) (actual
time=687.136..687.137 rows=0 loops=1)
   Recheck Cond: ((i2 = 31286) AND (i5 = 209009) AND (i7 = 653944) AND (i9 = 256927)
AND (i10 = 285508) AND (i12 = 93431))
   Rows Removed by Index Recheck: 1034
   Heap Blocks: exact=1033
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..1392164.00 rows=1 width=0) (actual
time=686.458..686.458 rows=1034 loops=1)
         Index Cond: ((i2 = 31286) AND (i5 = 209009) AND (i7 = 653944) AND (i9 =
256927) AND (i10 = 285508) AND (i12 = 93431))
 Planning Time: 0.140 ms
 Execution Time: 687.185 ms
```

Für Bloom-Filter-Indexe gelten die folgenden Einschränkungen:

- Operatorenklassen sind standardmäßig nur für Datentypen *int4* und *test* implementiert.
- Es sind nur Gleichheitsoperationen möglich.
- Bloom-Filter-Indexe können nicht als UNIQUE-Indexe eingesetzt werden.
- NULL-Werte in der WHERE-Bedingung werden nicht unterstützt.

Der Bloom-Filter-Index ist ein sehr gutes Beispiel für die Offenheit von PostgreSQL und kann als Blaupause für das Erstellen eigener Indextypen verwendet werden.

11.9 Indexe und Parallelität

Das Erstellen von Indexen kann mit Verwendung von mehreren CPUs erfolgen. Allerdings wird das aktuell nur für B-Tree-Indexe unterstützt. Wichtig für die Performance der Indexerstellung ist auch der Parameter *maintenance_work_mem*. Er spezifiziert die maximale Größe des Memory, das dafür benutzt werden kann. Ist er zu klein gewählt, werden möglicherweise weniger parallele Prozesse gestartet. Jeder parallele Prozess benötigt mindestens 32 Mbyte Memory.

TIPP: Die Indexerstellung, insbesondere wenn sie mit mehreren parallelen Prozessen erfolgt, kann zu einem hohen Ressourcenverbrauch des I/O-Subsystems führen. Um dies zu vermeiden und den Einfluss auf andere Sessions zu begrenzen, kann der Parameter *max_parallel_maintenance_workers* verwendet werden. PostgreSQL startet dann nicht mehr parallele Prozesse als vorgegeben. Der Parallelitätsgrad kann auch auf Tabellenebene begrenzt werden.

Um den Performance-Unterschied zu zeigen, wird der Index aus Listing 11.50 mit Parallelitätsgrad eins und vier erstellt.

Listing 11.54 Parallele Index-Erstellung

```
(postgres@localhost:5432)[hanser]> ALTER TABLE tbloom SET (parallel_workers=1);
ALTER TABLE
(postgres@localhost:5432)[hanser]> CREATE INDEX btreeidx
> ON tbloom (i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16);
CREATE INDEX
Time: 125190,541 ms (02:05,191)
(postgres@localhost:5432)[hanser]> ALTER TABLE tbloom SET (parallel_workers=4);
ALTER TABLE
(postgres@localhost:5432)[hanser]> CREATE INDEX btreeidx
> ON tbloom (i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15, i16);
CREATE INDEX
Time: 64468,577 ms (01:04,469)
```

Die Ausführungszeit wurde durch die Parallelisierung fast halbiert. Die Ausführungszeit skaliert nicht zu 100 Prozent mit der Anzahl der Prozesse, da immer noch I/O-Aktivitäten eingebunden sind und die parallelen Prozesse in einem seriellen Prozess zusammengeführt werden müssen.

Der Fortschritt der Indexerstellung kann über eine Abfrage der View *pg_stat_progress_index* verfolgt werden.

Listing 11.55 Fortschritt der Indexerstellung abfragen

```
(postgres@localhost:5432)[hanser]> SELECT command, phase, blocks_done, blocks_total
> FROM pg_stat_progress_create_index;
   command    |             phase              | blocks_done | blocks_total
--------------+--------------------------------+-------------+--------------
 CREATE INDEX | building index: scanning table |      267049 |       909091
```

Für Abfrage großer Indexe sind parallele Index-Scans ein wichtiges Element zur Performance-Verbesserung. Das betrifft sowohl parallele „Indexscans“ und parallele „Index-Only-Scans“. Aktuell werden parallele Indexscans nur für B-Tree-Indexe unterstützt. Ein Beispiel finden Sie in Listing 11.13.