13 Einsatz großer Datenbanken

Die durchschnittliche Größe von Datenbanken hat in den letzten Jahren deutlich zugenommen und der Trend für die Zukunft zeigt weiter nach oben. Datenbanksysteme sind der Schlüssel und die Basis aller Informationen. Es gibt kaum eine Applikation, die keine Datenbank verwendet. Die Möglichkeit der strukturierten und sicheren Ablage hat dazu geführt, dass kaum Daten außerhalb von Datenbanken gespeichert werden. Relationale Datenbanksysteme sind immer noch in der Überzahl, auch wenn sich die Art und Weise der Speicherung von Daten in der Zukunft ändern wird.

PostgreSQL ist in der Lage, große Datenbanken robust und performant zu verwalten. Für alle Systeme gilt, dass der Umgang mit sehr großen Datenbanken eigenen Gesetzen unterliegt und spezielle Features benötigt werden.

Die Planung für den Einsatz großer Datenbanken beginnt beim Design der Infrastruktur sowie der Applikationen. Häufig, aber nicht ausschließlich, begegnet man großen Datenbanken im Data Warehouse-Umfeld.

Das ständige Wachstum des Datenbestands sowie der Anzahl von konkurrierenden Benutzern stellt Hersteller, Designer und Administratoren immer wieder vor neue Herausforderungen. Dazu kommt der Umstand, dass die Hardwarekomponenten in den letzten Jahren kaum schneller geworden sind. Die Taktfrequenz von CPU-Kernen stagniert seit Jahren. Im Zuge der „Green IT“ ist man eher dazu übergegangen, die Single CPU Clockspeed zu reduzieren, um den Energieverbrauch und die Wärmeentwicklung zu senken. Bei den I/O-Subsystemen drängen immer mehr Solid State Disks auf den Markt, die einen besseren Durchsatz gegenüber Disk-Spindeln haben, aber auch einige Nachteile mit sich bringen.

Um große Datenmengen in vertretbare Zeit verarbeiten zu können, ist man dazu übergegangen, intelligente Softwarelösungen zu entwickeln, die Systeme größer zu machen und stark zu parallelisieren.

Die Firma Oracle hat die Oracle Database Machine (ODMExadata) auf den Markt gebracht. Diese enthält unter vielen anderen Lösungen Smart Scans und Storage-Indexe, die physikalische I/O-Operationen stark reduzieren und eine virtuelle Durchsatzrate von vielen Gigabyte pro Sekunde ermöglichen. In-Memory-Datenbanksysteme sind populär, wenngleich noch teuer, da in der Regel die gesamte Datenbank in den Memory passen muss.

Features für die Parallelisierung von Prozessen und Operationen sind eine notwendige Voraussetzung für den Einsatz großer Datenbanken. Mit dem Wachstum des Datenbestands ist die Erhöhung des Parallelitätsgrads eine gute Möglichkeit für die Verbesserung der Skalierbarkeit.

PostgreSQL war in dieser Hinsicht nicht untätig und hat eine ganze Reihe neuer Features eingeführt. Zu den wichtigsten gehören:

- Partitionierung: Einführung des Native Partitioning in PostgreSQL 10.
- Paralleles SQL: Ständige Erweiterung der Funktionalität für paralleles SQL in den Versionen 9 und 10.
- Materialized Views: Möglichkeit, mit aggregierten Daten zu arbeiten.
- BRIN-Indexe: Intelligente Lösung, um große Datenmengen zu scannen.

13.1 Partitionierung von Tabellen

Als bisherige Methode zur Partitionierung von Tabellen wurde die Funktionalität „Table Inheritance“ verwendet. Dabei wurden mehrere Tochtertabellen angelegt und die Konsistenz mithilfe von Check Constraints und Triggern gewährleistet. Diese Technologie hat einige entscheidende Nachteile. So müssen zum Beispiel INSERT-Anweisungen über Trigger an die Tochtertabellen weitergegeben werden, was klare Performancenachteile mit sich bringt. Mit der Version 10 wurde das „Native Table Partitioning“ eingeführt.

13.1.1 Native Table Partitioning

Die Technologie von PostgreSQL 10 ist wesentlich performanter und verwendet eine effiziente Verteilungsmethode auf die Tochtertabellen. Es müssen keine Constraints und Trigger mehr gebildet werden. Das verbessert auch die Zuverlässigkeit und bietet zusätzliche Verwaltungsoptionen.

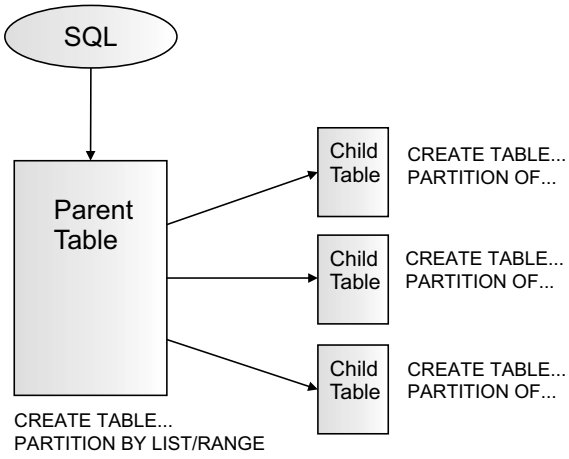


Bild 13.1 Native Table Partitioning

Es stehen zwei Arten zur Verfügung: List Partitioning und Range Partitioning. Beim List Partitioning werden den einzelnen Partitionen bestimmte Werte zugeordnet. Es gibt die Einschränkung, dass die Partitionierung nur auf Basis einer Spalte erfolgen kann. Zunächst wird die Elterntabelle mit der PARTITION BY-Klausel angelegt. Die Tabelle ist noch nicht funktionsfähig, es können keine Sätze eingefügt werden. Im nächsten Schritt werden die Tochtertabellen angelegt. Darin werden letztendlich die Daten gespeichert. Die Elterntabelle dient der Kommunikation mit der SQL-Anweisung sowie dem Routing in die einzelnen Partitionen. In Listing 13.1 finden Sie ein Beispiel für eine partitionierte Tabelle mit List Partitioning.

Listing 13.1 Eine Tabelle mit List Partitioning anlegen

```
postgres@[local]:5432)[postgres]> CREATE TABLE employees (
 > employee_id INT,
 > department_id INT,
 > first_name VARCHAR(30),
 > last_name VARCHAR(30),
 > hr_id INT,
 > salary INT)
 > PARTITION BY LIST(department_id);
CREATE TABLE
(postgres@[local]:5432)[postgres > CREATE TABLE employees_p10
 > PARTITION OF employees
 > FOR VALUES IN (10);
CREATE TABLE
. . .
```

In „psql“ kann die Definition der Elterntabelle einschließlich der zugehörigen Partitionen angezeigt werden (siehe Listing 13.2). Eine partitionierte Tabelle kann über die Spalte „relkind“ im View „pg_class“ identifiziert werden.

Listing 13.2 Definition einer partitionierten Tabelle anzeigen

```
(postgres@localhost:5432)[postgres]> \d+ employees
                  Table "public.employees"
    Column     |         Type          | Collation | Nullable
---------------+-----------------------+-----------+----------
 employee_id   | integer               |           |
 department_id | integer               |           |
 first_name    | character varying(30) |           |
 last_name     | character varying(30) |           |
 hr_id         | integer               |           |
 salary        | integer               |           |
Partition key: LIST (department_id)
Partitions: employees_p10 FOR VALUES IN (10),
            employees_p20 FOR VALUES IN (20),
            employees_p30 FOR VALUES IN (30)
(postgres@localhost:5432)[postgres]> SELECT relname, relpages, relkind
> FROM pg_class WHERE relname like 'employees%';
    relname    | relpages | relkind
---------------+----------+---------
 employees     |        0 | p
 employees_p10 |    12346 | r
 employees_p20 |    12346 | r
 employees_p30 |    12346 | r
```

HINWEIS: Beim List Partitioning ist zu beachten, dass für alle potenziellen Werte des Partitionsschlüssels eine Partition existieren muss. Andernfalls kommt es zum Fehler „ERROR: no partition of relation „employees“ found for row“.

Für das Range Partitioning können Bereiche, also Minimal- und Maximalwert für den Partitionsschlüssel, angegeben werden. Im Gegensatz zum List Partitioning können mehrere Spalten als Partitionsschlüssel angegeben werden. Das Anlegen ist analog zum List Partitioning. Zuerst muss die Elterntabelle und danach die partitionierten Tabellen angelegt werden (siehe Beispiel in Listing 13.3).

Listing 13.3 Eine Tabelle mit Range Partitioning anlegen

```
(postgres@localhost:5432)[postgres]> CREATE TABLE sales(
>   sales_id      INT,
>   customer_id   INT,
>   product_id    INT,
>   sales_date    DATE,
> amount        NUMERIC(12,2))
> PARTITION BY RANGE (sales_date);
CREATE TABLE
(postgres@localhost:5432)[postgres]> CREATE TABLE sales_2017_01
> PARTITION OF sales
> FOR VALUES FROM ('2017-01-01') TO ('2017-01-31');
CREATE TABLE
(postgres@localhost:5432)[postgres]> CREATE TABLE sales_2017_02
> PARTITION OF sales
> FOR VALUES FROM ('2017-02-01') TO ('2017-02-28');
CREATE TABLE
. . .
(postgres@localhost:5432)[postgres]> \d+ sales
               Table "public.sales"
   Column    |     Type      | Collation | Nullable
-------------+---------------+-----------+----------
 sales_id    | integer       |           |
 customer_id | integer       |           |
 product_id  | integer       |           |
 sales_date  | date          |           |
 amount      | numeric(12,2) |           |
Partition key: RANGE (sales_date)
Partitions: sales_2017_01 FOR VALUES FROM ('2017-01-01') TO ('2017-01-31'),
            sales_2017_02 FOR VALUES FROM ('2017-02-01') TO ('2017-02-28')
```

Indexe können an den Tochtertabellen angelegt werden, jedoch nicht an der Elterntabelle. Es ist gestattet, aber nicht notwendig, einen Index für den Partitionsschlüssel anzulegen. Es gibt kein Werkzeug, um sicherzustellen, dass Indexe auf allen Tochtertabellen angelegt sind. Diese müssen individuell gewartet und kontrolliert werden.

Partitionen können mit einem DROP-Befehl gelöscht werden. Mit dieser Methode können zum Beispiel historische Daten sehr effizient und sicher entfernt werden:

```
postgres@localhost:5432)[postgres]> DROP TABLE sales_2017_01;
DROP TABLE
```

Sollen die Daten noch erhalten bleiben, jedoch nicht mehr in der Elterntabelle auftauchen, dann kann eine Partition mit dem DETACH-Befehl entfernt werden (siehe Listing 13.4).

Listing 13.4 Eine Partition von der Tabelle entfernen

```
(postgres@localhost:5432)[postgres]> SELECT count(*) FROM employees;
  count
---------
 3000000
(postgres@localhost:5432)[postgres]> ALTER TABLE employees DETACH PARTITION
employees_p30;
ALTER TABLE
(postgres@localhost:5432)[postgres]> SELECT count(*) FROM employees;
  count
---------
 2000000
(postgres@localhost:5432) postgres]> SELECT count(*) FROM employees_p30;
  count
---------
 1000000
```

Für das Native Table Partitioning gelten die folgenden Einschränkungen:

- Primärschlüssel werden auf partitionierten Tabellen nicht unterstützt. Das hat zur Folge, dass auch keine Fremdschlüssel auf anderen Tabellen, die auf die partitionierte Tabelle verweisen, angelegt werden können.
- Ein Update auf den Partitionsschlüssel, der das Umschichten des Satzes in eine andere Partition zur Folge hat, bricht ab mit der folgenden Fehlermeldung: „ERROR: new row for relation“ sales_2017_01 „violates partition constraint“.
- Trigger müssen auf den Tochtertabellen und können nicht zentral auf der Elterntabelle definiert werden.

HINWEIS: Falls Sie bereits Inheritance Partitioning einsetzen, dann sollten Sie die Tabellen auf Native Partitioning umstellen. Einerseits wird Inheritance Partitioning nicht weiterentwickelt werden und andererseits ist es sinnvoll, die Performancevorteile von Native Partitioning zu nutzen. Hinweise zur Migration auf Native Partitioning finden Sie in Kapitel 3, „Upgrade auf Version 10“.

13.2 Paralleles SQL

Die Entscheidung, ob eine SQL-Anweisung oder deren Operationsstufen parallel ausgeführt werden, trifft der Query-Planer (Optimizer). Die folgenden Parameter beeinflussen die Entscheidung, ob parallele Ausführungspläne generiert werden können:

- *max_parallel_workers_per_gather:* Legt die maximal Anzahl von Worker-Prozessen fest, die für einen einzelnen Ausführungsschritt eines Query-Plans gestartet werden können. Der Standardwert ist 2. Diese Prozesse werden auch vom Pool, der durch den Parameter „max_worker_processes“ begrenzt ist, abgezogen. Eine weitere Begrenzung stellt der übergeordnete Wert „max_worker-processes“ dar.

- *dynamic_shared_memory_type:* Darf nicht auf den Wert „none“ gesetzt werden, damit paralleles SQL ausgeführt wird. Die parallelen Prozesse benötigen Dynamic Shared Memory, um Daten gegenseitig auszutauschen.

Derr EXPLAIN-Befehl gibt mit dem Ausführungsplan die Anzahl von parallelen Worker-Prozessen, die der Query-Planer bestimmt hat, zurück. Im Beispiel in Listing 13.5 hat sich der Query-Planer für zwei Worker-Prozesse entschieden, obwohl die Hardware mehr Systemressourcen zur Verfügung hat. Die Begrenzung kommt vom Parameter „max_parallel_workers_per_gather“, der standardmäßig auf dem Wert 2 steht.

Listing 13.5 Paralleles SQL auf einer großen Tabelle

```
(postgres@[local]:5432)[hanser]> EXPLAIN
SELECT count(*) FROM big
WHERE cont LIKE '%text%';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=591117.23..591117.24 rows=1 width=8)
   ->  Gather  (cost=591117.01..591117.22 rows=2 width=8)
       Workers Planned: 2
       ->  Partial Aggregate  (cost=590117.01..590117.02 rows=1 width=8)
          ->  Parallel Seq Scan on big  (cost=0.00..569283.68 rows=8333334 width=0)
                    Filter: ((cont)::text ~~ '%text%'::text)
```

Nach einer Erhöhung des Parameterwertes auf 8 wählt der Optimizer sechs parallele Worker-Prozesse (siehe Listing 13.6). Die Formulierung „Workers Planned“ im Ausführungsplan ist durchaus zutreffend. Es handelt sich um eine Einschätzung des Optimizers und Berücksichtigung der begrenzenden Parameter. Wenn während der Ausführung nicht genügend parallele Prozesse mehr zur Verfügung stehen, werden entsprechend weniger gestartet. Eine solche Begrenzung kann durch die Werte der Parameter „max_worker_processes“ oder „max_parallel_workers“ ausgelöst sein.

Listing 13.6 Den Parallelitätsgrad einer SQL-Abfrage erhöhen

```
(postgres@[local]:5432)[hanser]> EXPLAIN
SELECT count(*) FROM big
WHERE cont LIKE '%text%';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=516117.63..516117.64 rows=1 width=8)
   ->  Gather  (cost=516117.01..516117.62 rows=6 width=8)
       Workers Planned: 6
       ->  Partial Aggregate  (cost=515117.01..515117.02 rows=1 width=8)
         ->  Parallel Seq Scan on big  (cost=0.00..506783.67 rows=3333334 width=0)
                    Filter: ((cont)::text ~~ '%text%'::text)
```

Im vorliegenden Beispiel wurden folgenden Antwortzeiten erzielt:

- Parallel mit 6 Worker-Prozessen: `Time: 906.751 ms`
- Nicht parallel: `Time: 2957.980 ms (00:02.958)`

Mit sechs Worker-Prozessen hat sich die Ausführungszeit in etwa gedrittelt. Paralleles SQL skaliert also nicht linear. Dies liegt in der Natur der parallelen Verarbeitung. Jeder parallelisierte Prozess unterteilt sich in einen Anteil von parallelen Teilen und Teilen, die nicht parallelisiert werden können und damit sequenziell laufen (siehe Bild 13.2).

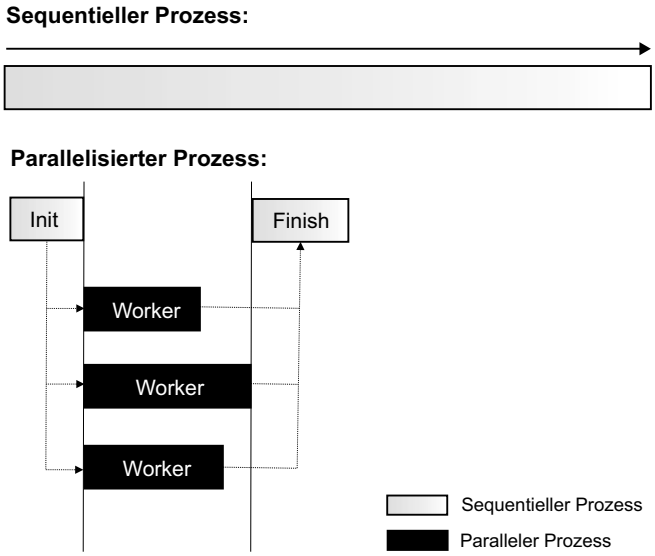


Bild 13.2 Das Prinzip parallelisierter Prozesse

Zu den sequenziellen Anteilen eines parallelisierten Prozesses gehören mindestens ein Initial- und ein Finish-Prozess. Im Fall eines parallelen Scans einer Tabelle muss die Arbeit an die parallelen Prozesse verteilt und die Ergebnisse müssen nach dem Scan wieder zusammengeführt werden.

TIPP: Die Parameter zur Begrenzung der Anzahl von parallelen Prozessen schützen das System vor Überlastung. Im Data Warehouse-Umfeld ist eine CPU-Auslastung von 100 Prozent prinzipiell kein Problem. Der Server muss jedoch vor einer Überlastung geschützt werden. Wird die Run Queue zu lang, gehen die Antwortzeiten für alle Prozesse in den Keller.

PostgreSQL ist in der Lage, die folgenden Aufgaben zu parallelisieren:

- Parallele Scans
- Parallele Joins
- Parallele Aggregation

Die folgenden Scan-Operationen können in PostgreSQL 10 parallel ausgeführt werden:

- *Parallel Sequential Scan:* Die Blöcke einer Tabelle werden auf die Worker-Prozesse verteilt.
- *Parallel Bitmap Heap Scan:* Der Master-Prozess durchsucht den Index und erstellt eine Bitmap für die Blöcke, die gelesen werden müssen. Diese werden auf die Worker-Prozesse aufgeteilt und parallel gescannt.
- *Parallel Index Scan:* Die Arbeit wird auf die Worker-Prozesse verteilt. Das Durchsuchen der Indexteile erfolgt dann parallel. Die Worker-Prozesse liefern die zugehörigen Tuples an den Master-Prozess.
- *Parallel Index-Only Scan:* Funktioniert wie ein Parallel Index Scan. Es werden keine Tupples gelesen, da sich alle Informationen im Index selbst befinden.

Parallele Joins sind seit der Version 10 für alle drei Methoden – Nested Loops, Hash Joins und Merge Joins – möglich. Alle Join-Methoden benutzen einen inneren Loop, die sogenannte Driving Table. Der innere Loop kann auch das Ergebnis eines vorangegangenen Joins sein. Jeder Worker-Prozess durchläuft dabei den ganzen inneren Loop.

Eine parallele Aggregation erfolgt in zwei Etappen. Jeder der parallelen Worker-Prozesse führt die Aggregation für seinen Bereich durch. Im zweiten Schritt werden die Ergebnisse an den Master-Prozess übergeben. Der Master-Prozess bringt die Ergebnisse zusammen.

Aktuell gibt es folgende Einschränkungen für die Ausführung von parallelem SQL:

- Die SQL-Anweisung schreibt Daten oder sperrt Tabellen oder Datensätze.
- Die SQL-Anweisung benutzt Funktionen, die als „PARALLEL UNSAFE“ markiert sind.
- Das SQL läuft innerhalb einer Anweisung, die bereits parallel ausgeführt wird.

Auch wenn der Query Optimizer einen parallelen Plan vorgegeben hat, kann es passieren, dass die Ausführung nicht parallel erfolgt. Diese Situation kann eintreten, wenn zum Beispiel die maximale Anzahl von Worker-Prozessen erreicht ist.

Parallel Query funktioniert seit Version 10 auch für Prepare- und Execute Anweisungen, Bitmap Joins und Merge Joins. Weiterhin können Index Scans auch parallelisiert werden. Die Entscheidung, ob SQL-Abfragen parallel ausgeführt werden, trifft der Query Optimizer. In Listing 13.7 finden Sie ein Beispiel für Prepare und Execute.

HINWEIS: Tabellen müssen nicht zwangsläufig partitioniert sein. Paralleles SQL funktioniert auch mit nicht partitionierten Tabellen.

Listing 13.7 Paralleles SQL mit Prepare und Execute

```
(postgres@localhost:5432)[postgres]> PREPARE c1 AS
SELECT entry_date,count(*) FROM order_entry GROUP BY entry_date;
PREPARE
(postgres@localhost:5432)[postgres]> EXPLAIN EXECUTE c1;
                                    QUERY PLAN
-------------------------------------------------------------------------------
Finalize GroupAggregate  (cost=76779.30..76784.30 rows=200 width=12)
Group Key: order_entry_201201.entry_date
-> Sort  (cost=76779.30..76780.30 rows=400 width=12)
   Sort Key: order_entry_201201.entry_date
-> Gather  (cost=76720.01..76762.01 rows=400 width=12)
   Workers Planned: 2
-> Partial HashAggregate  (cost=75720.01..75722.01 rows=200 width=12)
   Group Key: order_entry_201601.entry_date
-> Append  (cost=0.00..63220.00 rows=2500002 width=4)
-> Parallel Seq Scan on order_entry_201201 (cost=0.00..10536.67 rows=416667 width=4)
-> Parallel Seq Scan on order_entry_201202 (cost=0.00..10536.67 rows=416667 width=4)
-> Parallel Seq Scan on order_entry_201203 (cost=0.00..10536.67 rows=416667 width=4)
-> Parallel Seq Scan on order_entry_201204 (cost=0.00..10536.67 rows=416667 width=4)
-> Parallel Seq Scan on order_entry_201205 (cost=0.00..10536.67 rows=416667 width=4)
-> Parallel Seq Scan on order_entry_201206 (cost=0.00..10536.67 rows=416667 width=4)
```

Das Beispiel in Listing 13.8 zeigt einen parallelen Index-Only Scan für eine nicht partitionierte Tabelle. Im Data Warehouse-Umfeld werden häufig Index-Only Scans benötigt. PostgreSQL 10 unterstützt nun auch da parallele Abfragen.

Listing 13.8 Paralleler Index-Only Scan

```
(postgres@localhost:5432)[postgres]> \d+ sales
          Table "public.sales"
  Column     |           Type           |
-------------+--------------------------+--
 sales_id    | bigint                   |
 customer_id | integer                  |
 product_id  | integer                  |
 sales_date  | timestamp with time zone |
 amount      | numeric                  |
Indexes:
    "pk_sales" PRIMARY KEY, btree (sales_
    "idx_sales" btree (product_id)
(postgres@localhost:5432)[postgres]> EXPLAIN
> SELECT count(*) FROM sales WHERE sales_id > 10 AND sales_id < 400000;
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=17945.78..17945.79 rows=1 width=8)
   ->  Gather  (cost=17945.57..17945.78 rows=2 width=8)
         Workers Planned: 4
         ->  Partial Aggregate  (cost=16945.57..16945.58 rows=1 width=8)
               ->  Parallel Index Only Scan using pk_sales on sales
(cost=0.44..16419.32 rows=210497 width=0)
                     Index Cond: ((sales_id > 10) AND (sales_id < 400000))
```

Um die Arbeit zwischen den Worker-Prozessen parallelisieren zu können, muss die Driving Table für die parallele Weiterverarbeitung aufteilbar sein (siehe Listing 13.9). In diesem Beispiel wird ein serieller Bitmap Index Scan ausgeführt. Dabei wird eine Datenstruktur im Shared Memory mit allen Blöcken aufgebaut, die gescannt werden müssen. Die Worker-Prozesse können dann den Heap Scan parallel ausführen.

Listing 13.9 Paralleler Bitmap Heap Scan

```
(postgres@localhost:5432)[postgres]> EXPLAIN
> SELECT count(*) FROM sales
> WHERE product_id < 100
> GROUP BY product_id;
                                QUERY PLAN
-------------------------------------------------------------------------------
Finalize GroupAggregate  (cost=251653.96..251666.46 rows=500 width=12)
  Group Key: product_id
  ->  Sort  (cost=251653.96..251656.46 rows=1000 width=12)
      Sort Key: product_id
  ->  Gather  (cost=251499.14..251604.14 rows=1000 width=12)
      Workers Planned: 2
      ->  Partial HashAggregate  (cost=250499.14..250504.14 rows=500 width=12)
          Group Key: product_id
      ->  Parallel Bitmap Heap Scan on sales  (cost=74441.67..242213.86 rows=1657055
width=4)
          Recheck Cond: (product_id < 100)
      ->  Bitmap Index Scan on idx_sales  (cost=0.00..73447.43 rows=3976933 width=0)
          Index Cond: (product_id < 100)
```

13.3 Materialized Views

Materialized Views sind eine wichtige Voraussetzung, um in großen Datenbanken die erwartete Performance erzielen zu können. Aggregierte Tabellen können nicht nur Ergebnisse in wesentlich kürzerer Zeit liefern, sondern schonen auch die Systemressourcen. In Data Warehouse-Datenbanken sind aggregierte Tabellen häufig zu finden, aber nicht darauf beschränkt.

Auch Materialized Views verwenden das Rule System wie normale Views, allerdings werden die Ergebnisdaten in einer Tabelle gespeichert. In Listing 13.10 wird eine große Tabelle mit 50 Millionen Sätzen angelegt, so wie sie in einem Data Warehouse vorkommt.

Listing 13.10 Eine große Tabelle anlegen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE sales (
>   sales_id      INT,
>   customer_id   INT,
>   product_id    INT,
>   sales_date    DATE,
> amount        NUMERIC(12,2));
(postgres@localhost:5432)[hanser]> INSERT INTO sales
> SELECT n, MOD(n,100) + 1, MOD(n,10) + 1,
> TIMESTAMP '2016-01-01 00:00:00' + RANDOM() * (now() - TIMESTAMP '2016-01-01
00:00:00'),
> RANDOM()::NUMERIC * 1000
> FROM generate_series(1,50000000) x(n);
INSERT 0 5000000
```

Um die Abfragen auf die Verkaufszahlen pro Produkt zu beschleunigen, wird in Listing 13.11 ein Materialized View erstellt.

Listing 13.11 Ein Materialized View erstellen

```
(postgres@localhost:5432)[hanser]> CREATE MATERIALIZED VIEW sales_sum
> AS SELECT product_id, SUM(amount) FROM sales
> GROUP BY product_id;
SELECT 10
(postgres@localhost:5432)[hanser]> SELECT * FROM sales_sum ORDER BY 1;
 product_id |      sum
------------+---------------
          1 | 2498948596.70
          2 | 2500118594.65
          3 | 2499927067.36
          4 | 2499631272.60
          5 | 2500590545.55
          6 | 2499671040.93
          7 | 2501213310.02
          8 | 2500211103.53
          9 | 2500597263.13
         10 | 2500607800.36
(10 rows)
```

In Listing 13.12 finden Sie einen Laufzeitvergleich zwischen der Abfrage auf das Materialized View und der Originaltabelle mit 50 Millionen Sätzen. Während das Scannen der 50 Millionen Sätze 5 Minuten dauert, liefert das Materialized View das Ergebnis in weniger als einer Sekunde. Die Messung ist nach dem Start des Servers erfolgt. Wenn sich die Daten im Cache befinden, kann die Laufzeit variieren.

Listing 13.12 Lauzeitvergleich Materialized View und Tabelle

```
(postgres@localhost:5432)[hanser]> SELECT product_id, SUM(amount)
> FROM sales
> GROUP BY product_id ORDER BY 1;
product_id |      sum
-----------+--------------
         1 | 2498948596.70
. . .
Time: 5026,761 ms (00:05,027)
(postgres@localhost:5432)[hanser]>  SELECT * FROM sales_sum
> ORDER BY 1;
 product_id |      sum
------------+--------------
          1 | 2498948596.70
. . .
Time: 154,807 ms
```

Leider gibt es keinen automatischen Refresh-Mechanismus für Materialized Views. Der Refresh-Befehl (Listing 13.13) muss manuell abgesetzt oder in einen Job eingebunden werden. Dabei wird der Inhalt komplett ersetzt. Ein inkrementeller Refresh (Fast Refresh) ist nicht möglich. Mit der Option „CONCURRENTLY“ können SQL-Anweisungen auf das View zugreifen, während der Refresh läuft. Der Refresh läuft länger, allerdings muss die Session mit der Abfrage nicht auf das Beenden der Refreshs warten. Voraussetzung für einen konkurrierenden Refresh ist, dass das Materialized View einen eindeutigen Index besitzt.

Listing 13.13 Ein Materialized View aktualisieren

```
(postgres@localhost:5432)[hanser]> REFRESH MATERIALIZED VIEW sales_sum;
REFRESH MATERIALIZED VIEW
(postgres@localhost:5432)[hanser]> CREATE UNIQUE INDEX'
> i_sales_sum ON sales_sum(product_id);
CREATE INDEX
(postgres@localhost:5432)[hanser]> \d sales_sum
         Materialized view "public.sales_sum"
   Column   |  Type   | Collation | Nullable | Default
------------+---------+-----------+----------+---------
 product_id | integer |           |          |
 sum        | numeric |           |          |
Indexes:
    "i_sales_sum" UNIQUE, btree (product_id)
(postgres@localhost:5432)[hanser]> REFRESH MATERIALIZED VIEW
> CONCURRENTLY sales_sum;
REFRESH MATERIALIZED VIEW
```

13.4 BRIN-Indexe

B-Tree-Indexe sind effizient, solange sie eine gewisse Größe nicht überschreiten. Sie wachsen nahezu linear mit der Tabelle und werden für sehr große Tabellen langsam und ineffektiv. Große B-Tree-Indexe müssen häufig von der Disk nachgeladen werden und Index Scans laufen aufgrund der Tiefe der Verzweigungen vergleichsweise lang.

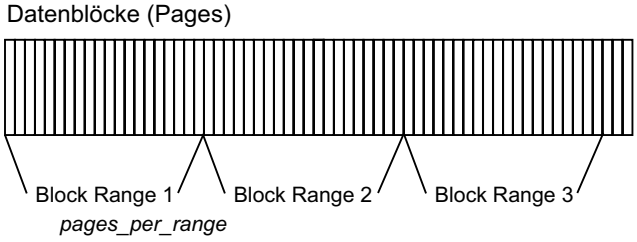
Seit der Version 9.5 gibt es Block Range-Indexe (BRIN). In einem Block werden nicht einzelne Einträge von Spaltenwerten gespeichert, sondern es werden Datenblöcke indiziert. Für jeden Block werden ein Minimal- und ein Maximalwert in Form von Bitmaps gespeichert (siehe Bild 13.3).

Ein BRIN-Index kann einen signifikanten Performancevorteil gegenüber einem B-Tree-Index erzielen. Der Aufbau des Index macht klar, wann er seine Stärken entfalten kann. Die besten Effekte werden erzielt, wenn die Werte der indizierten Spalte weitgehend in Sortierreihenfolge in den Blöcken gespeichert sind. Ein klassisches Beispiel ist das Auftragseingangsdatum in einer Auftragstabelle. Neue Aufträge werden in zeitlicher Reihenfolge aufgenommen und für gespeicherte Sätze ändert sich der Wert nicht.

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. Allerdings müssen möglicherweise zu viele Blöcke gescannt werden, wenn der Wert zu groß gewählt wird. Der Standardwert ist 128.

Der entscheidende Vorteil eines BRIN liegt in der Größe. Gerade für sehr große Tabellen ist der Unterschied gewaltig. Eines der Probleme von BRIN-Indexen ist, dass sich bei stark ändernden Tabellen die Qualität und die Effektivität reduzieren. Deshalb kommen BRIN-Indexe insbesondere im Data Warehouse-Umfeld zum Einsatz, wo Daten in einer bestimmten Reihenfolge geladen werden und Tabellen wenigen Veränderungen unterliegen.

BRIN-Indexe werden normal nicht verändert. Seit Version 10 gibt es eine automatische Pflege. Wird der Index mit dem Attribut „autosummarize=true“ angelegt, erfolgt eine Aktualisierung bei jedem manuellen und automatischen VACUUM-Lauf.



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 13.3 Block Range-Index (BRIN)

Für das folgende Beispiel wird in Listing 13.14 eine Tabelle mit mehr als 30 Millionen Sätzen angelegt.

Listing 13.14 Eine große Tabelle anlegen

```
(postgres@localhost:5432)[hanser]> CREATE TABLE temperature(
> location_id     INTEGER,
> t_time          TIMESTAMP,
> t_celsius       INTEGER);CREATE TABLE temperature(
CREATE TABLE
(postgres@localhost:5432)[hanser]> INSERT INTO temperature
> VALUES (1,generate_series('2017-01-01'::timestamp,'2017-12-31'::timestamp,'1
second'),
> round(random()*100)::int);
INSERT 0 31449601
```

Auswertungen erfolgen nach der Spalte „t_time“, in der sich Daten vom Typ „TIMESTAMP“ befinden. Zunächst wird ein normaler B-Tree-Index auf die Spalte gelegt (siehe Listing 13.15).

Listing 13.15 Einen B-Tree-Index erstellen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_temperature_btree
> ON temperature (t_time);
(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_temperature_btree" btree (t_time)
```

Der Ausführungsplan für eine Abfrage zeigt einen parallelen Index Scan auf den B-Tree-Index (siehe Listing 13.16). Die tatsächliche Ausführungszeit beträgt etwas mehr als 600 Millisekunden.

Listing 13.16 Paralleler Index Scan mit B-Tree-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT AVG(t_celsius)> FROM temperature
> WHERE t_time > '2017-02-28' AND t_time < '2017-04-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
Finalize Aggregate  (cost=95517.26..95517.27 rows=1 width=32) (actual
time=609.800..609.800 rows=1 loops=1)'
  ->  Gather  (cost=95517.05..95517.26 rows=2 width=32) (actual time=309.727..609.793
rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
  ->  Partial Aggregate  (cost=94517.05..94517.06 rows=1 width=32) (actual
time=285.748..285.748 rows=1 loops=3)
  ->  Parallel Index Scan using i_temperature_btree on temperature
(cost=0.56..91492.08 rows=1209986 width=4) (actual time=0.376..339.046 rows=92160
loops=3)
      Index Cond: ((t_time > '2017-02-28 00:00:00'::timestamp without time zone) AND
```

```
(t_time < '2017-04-01 00:00:00'::timestamp without time zone))
 Planning time: 0.504 ms
 Execution time: 858.128 ms
(postgres@localhost:5432)[hanser]> SELECT AVG(t_celsius)
> FROM temperature
> WHERE t_time > '2017-02-28' AND t_time < '2017-04-01';
         avg
---------------------
 49.9919256336536580
Time: 633,886 ms
```

Für den zweiten Test wird ein BRIN-Index mit einer Block Range von „128“ angelegt (siehe Listing 13.17).

Listing 13.17 Einen Block Range-Index anlegen

```
(postgres@localhost:5432)[hanser]> CREATE INDEX i_temperature
> ON temperature USING BRIN (t_time)
> WITH (pages_per_range = 128, autosummarize=true);
CREATE INDEX
(postgres@localhost:5432)[hanser]> ANALYZE temperature;
ANALYZE
(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_temperature" brin (t_time) WITH (pages_per_range='128', autosummarize='true')
```

Der Ausführungsplan in Listing 13.18 führt einen „Bitmap Index Scan“ durch, ebenfalls mit einem Parallelitätsgrad von 2. Die Kosten, verglichen mit dem Index Scan des B-Tree-Indexes in Listing 13.16, sind signifikant geringer. Die Ausführungszeit beträgt in etwa 268 Millisekunden.

Listing 13.18 Ausführungsplan mit Block Range-Index

```
(postgres@localhost:5432)[hanser]> EXPLAIN ANALYZE
> SELECT AVG(t_celsius)
> FROM temperature
> WHERE t_time > '2017-02-28' AND t_time < '2017-04-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------
Finalize Aggregate  (cost=223316.45..223316.46 rows=1 width=32) (actual
time=245.086..245.086 rows=1 loops=1)
   ->  Gather  (cost=223316.23..223316.44 rows=2 width=32) (actual
time=245.059..245.079 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
     ->  Partial Aggregate  (cost=222316.23..222316.24 rows=1 width=32) (actual
time=218.354..218.354 rows=1 loops=3)
     ->  Parallel Bitmap Heap Scan on temperature  (cost=763.63..219291.26
rows=1209986 width=4) (actual time=1.413..172.239 rows=921600 loops=3)
         Recheck Cond: ((t_time > '2017-02-28 00:00:00'::timestamp without time zone)
AND (t_time < '2017-04-01 00:00:00'::timestamp without time zone))
```

```
        Rows Removed by Index Recheck: 2816
        Heap Blocks: lossy=6897
        ->  Bitmap Index Scan on i_temperature  (cost=0.00..37.64 rows=2913861
width=0) (actual time=1.737..1.737 rows=176640 loops=1)
              Index Cond: ((t_time > '2017-02-28 00:00:00'::timestamp without time zone)
AND (t_time < '2017-04-01 00:00:00'::timestamp without time zone))
 Planning time: 0.183 ms
 Execution time: 279.457 ms
(postgres@localhost:5432)[hanser]> SELECT AVG(t_celsius)
> FROM temperature
> WHERE t_time > '2017-02-28' AND t_time < '2017-04-01';
         avg
---------------------
 49.9919256336536580
Time: 268,739 ms
```

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

- B-Tree-Index Scan: `actual time=0.376..239.046`
- BRIN-Index Scan: `actual time=1.737..1.737`

Das Beispiel ist natürlich ein Idealfall für den BRIN-Index. Die Daten wurden frisch geladen und zwar in Sortierung des Timestamps. Die Verteilung ist damit fast optimal. Beindruckend ist auch der Unterschied in der Größe der Indexe.

Listing 13.19 Größenvergleich B-Tree- und BRIN-Index

```
schema_name  |     index_name      | index_ratio | index_size | table_size
-------------+---------------------+-------------+------------+------------
 public      | i_temperature_btree |        0.43 | 674 MB     | 1565 MB
 public      | i_temperature       |           0 |  72 kB     | 1565 MB
```