Les index PostgreSQL

Et leur intégration avec Django

François FREITAG

Les emplois de l'inclusion
GIP Inclusion

26 août 2024

Sommaire

Les données, la requête

```sql
CREATE TABLE test1 (
    id INTEGER,
    content VARCHAR
);
```

```sql
SELECT content FROM test1 WHERE id = constant;
```

EXPLAIN ANALYZE

ANALYZE : Carry out the command and show actual run times and other statistics.

```
EXPLAIN ANALYZE SELECT content FROM test1 WHERE id = 12;

                     QUERY PLAN
--------------------------------------------------------------
 Seq Scan on test1  (cost=0.00..25.88 rows=6 width=32)
                    (actual time=0.016..0.017 rows=0 loops=1)
   Filter: (id = 12)
 Planning Time: 2.165 ms
 Execution Time: 0.098 ms
(4 rows)
```

Sommaire

Indexer les données

De la même manière qu'un livre contient souvent un index trié par ordre alphabétique, la base de données crée un index trié des données présentes dans une table.

```
CREATE INDEX test1_id_index ON test1 (id);
```

```
EXPLAIN ANALYZE SELECT content FROM test1 WHERE id = 12;

                         QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=4.20..13.67 rows=6 width=32)
                            (actual time=0.010..0.011 rows=0 loops=1)
   Recheck Cond: (id = 12)
   ->  Bitmap Index Scan on test1_id_index
               (cost=0.00..4.20 rows=6 width=0)
               (actual time=0.008..0.008 rows=0 loops=1)
         Index Cond: (id = 12)
 Planning Time: 0.118 ms
 Execution Time: 0.053 ms
(6 rows)
```

Les coûts d'un index

- Espace disque supplémentaire
- Maintenance de l'index lors des opérations en écriture (`INSERT`, `UPDATE`, `DELETE`)

Sommaire

B-Tree

- Index par défaut de PostgreSQL
- Equality (=), range queries (<, >, `BETWEEN`, `IN`), `NULL`

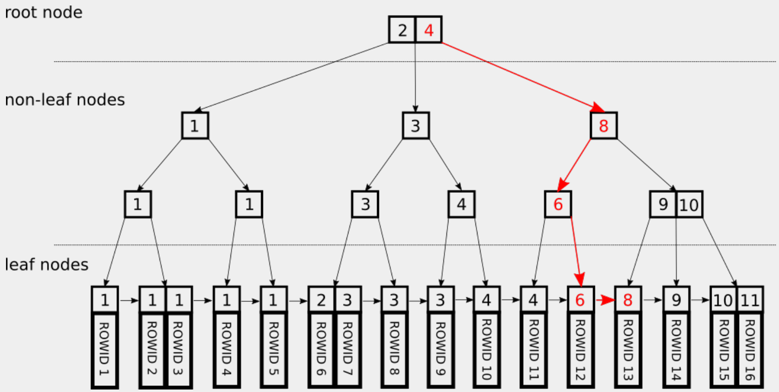
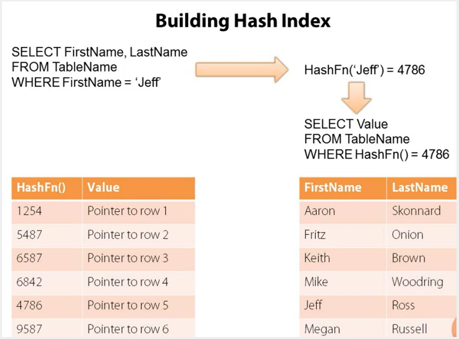


Figure – source : https://www.qwertee.io/blog/postgresql-b-tree-index-explained-part-1/

Hash

- Hash indexes store a 32-bit hash code derived from the value of the indexed column.
- Equality (=), not inequality.
- More compact and can be faster than a B-Tree index



Building Hash Index

```
SELECT FirstName, LastName
FROM TableName
WHERE FirstName = 'Jeff'
```

HashFn('Jeff') = 4786

```
SELECT Value
FROM TableName
WHERE HashFn() = 4786
```

HashFn()	Value
1254	Pointer to row 1
5487	Pointer to row 2
6587	Pointer to row 3
6842	Pointer to row 4
4786	Pointer to row 5
9587	Pointer to row 6

FirstName	LastName
Aaron	Skonnard
Fritz	Onion
Keith	Brown
Mike	Woodring
Jeff	Ross
Megan	Russell

Figure – source : https://gss-portal.com/knowledgebase/129/Hash-Index-Part-4.html/

GiST - Generalized Search Tree

- A balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes.
- Works on ranges (containment `<@`, overlap `&&`, strictly left/right «, », etc)
- Handles
  - ranges (integer, dates, IP subnets, etc),
  - geographical data,
  - string distance,
  - bioinformatics,
  - nearest neighbors,
  - and generally what domain experts can modelize with a set of standard operations.

SP-GiST - space-partitioned GiST

- Same as GiST, but allows for unbalanced structures.
- Can be much faster for domains where partitioning makes sense.

GIN - General INverted

- Appropriate for data values that contain multiple component values, such as arrays or JSON.
- Equality (=), containment (`<@`, `@>`), overlap (`&&`).

⇒ C'est ce que les `emails_email` utilisent, pour rechercher une adresse email dans la liste des champs `To`, `Cc` et `Bcc`.

BRIN - Block Range Index

- Designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table.
- e.g. storing a store's sale orders might have a date column on which each order was placed
- Lossy, must recheck the tuples and discarding those that do not match the query conditions
- Very small, avoids scanning large parts of the table

Bloom

- A Bloom filter is a space-efficient data structure that is used to test whether an element is a member of a set.
- Lossy, takes a signature based on the data and returns tuples matching the signature
- Many attributes and queries test arbitrary combinations of them
- Equality (=), not inequality.
- Requires an extension
- *Example on PostgreSQL documentation*

Sommaire

Functional index

```
CREATE INDEX test1_upper_content_index ON test1 (UPPER(content));
```

```
EXPLAIN ANALYZE SELECT content FROM test1 WHERE UPPER(content)='FOO';

                        QUERY PLAN
--------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=4.20..13.68 rows=6 width=32)
                            (actual time=0.010..0.011 rows=0 loops=1)
   Recheck Cond: (upper((content)::text) = 'FOO'::text)
   ->  Bitmap Index Scan on test1_upper_content_index
                            (cost=0.00..4.20 rows=6 width=0)
                            (actual time=0.008..0.008 rows=0 loops=1)
         Index Cond: (upper((content)::text) = 'FOO'::text)
 Planning Time: 0.142 ms
 Execution Time: 0.055 ms
```

Functional index

⚠️ Requires a functional query.

```
EXPLAIN ANALYZE SELECT content FROM test1 WHERE content='FOO';

                       QUERY PLAN
------------------------------------------------------
 Seq Scan on test1  (cost=0.00..25.88 rows=6 width=32)
                    (actual time=0.017..0.018 rows=0 loops=1)
   Filter: ((content)::text = 'FOO'::text)
 Planning Time: 0.865 ms
 Execution Time: 0.105 ms
(4 rows)
```

CIEmailField

Module `citext` pour les adresses e-mail des emplois de linclusion.

- Comparaison non sensible à la casse
- Pas besoin dappeler de fonction sur les valeurs
- Fonctionne avec `UNIQUE`
- Les emails nont pas besoin dune normalisation de l'unicode, car les emplois ne supportent pas l'*Email Address Internationalization*

Refs ce commit

Partial index

```
CREATE INDEX test1_large_ids_index ON test1 (id) WHERE id >= 300000;
```

```
EXPLAIN ANALYZE SELECT * FROM test1 WHERE id > 30000;

                   QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=7.43..22.72 rows=423 width=36)
                            (actual time=0.002..0.003 rows=0 loops=1)
   Recheck Cond: (id > 3000000)
   ->  Bitmap Index Scan on test1_composite_index
                            (cost=0.00..7.32 rows=423 width=0)
                            (actual time=0.001..0.002 rows=0 loops=1)
         Index Cond: (id > 30000)
 Planning Time: 0.047 ms
 Execution Time: 0.016 ms
(6 rows)
```

Composite index

```
CREATE INDEX test1_composite_index ON test1 (id, content);
```

```
EXPLAIN ANALYZE SELECT * FROM test1 WHERE id > 30000;

                   QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=7.43..22.72 rows=423 width=36)
                            (actual time=0.002..0.003 rows=0 loops=1)
   Recheck Cond: (id > 3000000)
   ->  Bitmap Index Scan on test1_composite_index
                            (cost=0.00..7.32 rows=423 width=0)
                            (actual time=0.001..0.002 rows=0 loops=1)
         Index Cond: (id > 30000)
 Planning Time: 0.047 ms
 Execution Time: 0.016 ms
(6 rows)
```

Most specialized index

```
EXPLAIN ANALYZE SELECT * FROM test1 WHERE id > 300000;

                            QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on test1  (cost=4.24..19.53 rows=423 width=36)
                            (actual time=0.011..0.012 rows=0 loops=1)
   Recheck Cond: (id > 3000000)
   ->  Bitmap Index Scan on test1_large_ids_index
                            (cost=0.00..4.13 rows=423 width=0)
                            (actual time=0.008..0.008 rows=0 loops=1)
         Index Cond: (id > 3000000)
 Planning Time: 0.178 ms
 Execution Time: 0.047 ms
(6 rows)
```

Sommaire

Django

```python
class Index(*expressions, fields=(), name=None, db_tablespace=None,
            opclasses=(), condition=None, include=None)

# Un index fonctionnel ?
```

Django

```python
class Index(*expressions, fields=(), name=None, db_tablespace=None,
            opclasses=(), condition=None, include=None)

# Un index fonctionnel ?
Index(Lower("title").desc(), name="lower_title_idx")
# Un index composite ?
```

```
class Index(*expressions, fields=(), name=None, db_tablespace=None,
            opclasses=(), condition=None, include=None)

# Un index fonctionnel ?
Index(Lower("title").desc(), name="lower_title_idx")
# Un index composite ?
Index("title", "pub_date", name="title_date_idx")
# Un index partiel ?
```

Django

```python
class Index(*expressions, fields=(), name=None, db_tablespace=None,
            opclasses=(), condition=None, include=None)

# Un index fonctionnel ?
Index(Lower("title").desc(), name="lower_title_idx")
# Un index composite ?
Index("title", "pub_date", name="title_date_idx")
# Un index partiel ?
Index(
      "pub_date",
      condition=Q(pub_date__lt=datetime.date(2020, 1, 1)),
      name="published_before_2020_idx",
)
```

Django

```python
# django.contrib.postgres.indexes:
class BloomIndex(*expressions, length=None, columns=(), **opts): ...
class BrinIndex(*expressions, autosummarize=None,
                pages_per_range=None, **opts): ...
class BTreeIndex(*expressions, fillfactor=None, **opts): ...
class GinIndex(*expressions, fastupdate=None,
               gin_pending_list_limit=None, **opts): ...
class GistIndex(*expressions, buffering=None,
                fillfactor=None, **opts): ...
class HashIndex(*expressions, fillfactor=None, **opts): ...
class SpGistIndex(*expressions, fillfactor=None, **opts): ...
```

Merci de votre attention

Avez-vous des questions ?