PostgreSQL Full Text Lecture Notes

In this lecture we will explore how PostgreSQL indexes work and how we build indexes for large text fields that contain natural language and how we can look into those fields and use indexes to search large text fields efficiently.

Additional Materials

- PowerPoint slides of the diagrams
- Sample SQL commands for this lecture
- URL for these notes: https://www.pg4e.com/lectures/05-FullText

Row Data Layout 📖 🏠 ⏩

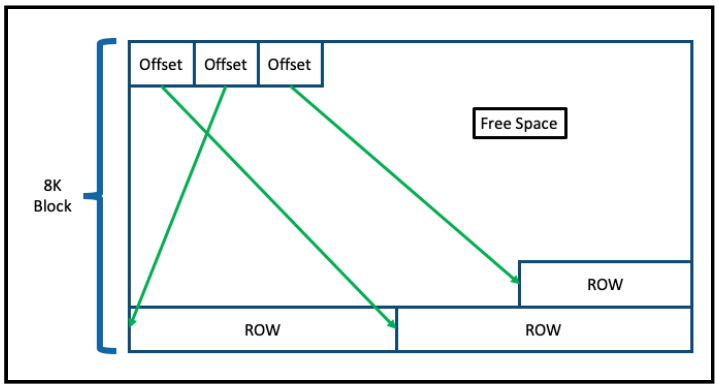
Rows can vary quite a bit in terms of length.

```
CREATE TABLE messages
 (id SERIAL,                -- 4 bytes
  email TEXT,               -- 10-20 bytes
  sent_at TIMESTAMPTZ,      -- 8 bytes
  subject TEXT,             -- 10-100 bytes
  headers TEXT,             -- 500-1000 bytes
  body TEXT                 -- 50-2000 bytes
                            -- 600-2500 bytes
);
```

Since modifying data is so important to databases, we don't pack store one row after another in a file. We arrange the file into blocks (default 8K) and pack the rows into blocks leaving some free space to make inserts updates, or deletes possible without needing to rewrite a large file to move things up or down.

PostgreSQL Organizes Rows into Blocks

- We read an entire block into memory (i.e. not just one row)
- Easy to compute the start of a block within a file for random access
- There are the unit of caching in memory
- They are (often) the unit of locking when we think we are locking a row



What is the Best Block Size?

- Blocks that are small waste free space / fragmentation
- Large blocks take more memory in cache be cached for a given memory size
- Large blocks longer to read and write to/from SSD

If we have a table that contains 1GB (125,000 blocks) of data, a sequential scan from a fast SSD takes about 2 seconds while with careful optimization, reading a random block can be fast as 1/50000th of a second. Some SSD drives can read as many as 32 different random blocks in a single read request. If the block is already cached in memory it is even faster. Sequential scans are very bad.

References

- The File Layout of Postgres Tables (Blog Post)
- Introduction to PostgreSQL physical storage (Blog Post)

- PostgreSQL on SSD - 4kB or 8kB pages? (Blog Post)

Indexes ⏪ 📖 🏠 ⏩

Assume each row in the **users** table is about 1K, we could save a lot of time if somehow we had a hint about which row was in which block.

```
email              | block
-------------------+------
anthony@umich.edu  | 20175
csev@umich.edu     | 14242
colleen@umich.edu  | 21456

SELECT name FROM users WHERE email='csev@umich.edu';
SELECT name FROM users WHERE email='colleen@umich.edu';
SELECT name FROM users WHERE email='anthony@umich.edu';
```

Our index would be about 30 bytes per row which is much smaller than the actual row data. We store index data in 8K blocks as well - as indexes grow in size we need to find was to avoid reading the entire index to look up one key. We need an index to the index. For string logical keys, a B-Tree index is a good, general solution. B-Trees keep the keys in sorted order by reorganizing the tree as keys are inserted.

PostgreSQL Index Types

- B-Tree - The default for many applications - automatically balanced as it grows
- BRIN - Block Range Index - Smaller / faster if data is mostly sorted
- Hash - Quick lookup of long key strings
- GIN - Generalized Inverted Indexes - multiple values in a column
- GiST - Generalized Search Tree
- SP-GiST - Space Partitioned Generalized Search Tree

References

- PostgreSQL Index Types (Blog Post)
- B-Tree Index (WikiPedia)
- Block Range Index (WikiPedia)

Forward and Inverted Indexes ⏪ 📖 🏠 ⏩

It is not a perfect metaphor but in general there are two categories of indexes:

- **Forward indexes** - You give the index a logical key and it tells you where to find the row that contains the key. (B-Tree, BRIN, Hash)
- **Inverse indexes** - You give the index a string (query) and the index gives you a list of *all* the rows that match the query. (GIN, GiST)

The metaphor is not perfect - because B-tree indexes are stored in sorted order, if you give a B-Tree the prefix of a logical key, it can give you a set of rows...

The most typical use case for an **inverse index** is to quickly search text documents wit one or a few words.

References

- Inverted_indices (Wikipedia)

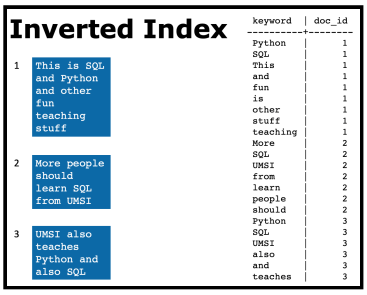
Similar to Google Search ⏪ 📖 🏠 ⏩

- Crawl: Retrieve documents, parse them and create an **inverted index**
- Search: Take keywords, find the documents with the words then rank them and present results

References

- Google I/O '08 Keynote by Marissa Mayer
- How Search Works - Matt Cutts

Inverted Indexes - Using only SQL ⏪ 📖 🏠 ⏩



We can split long text columns into space-delimited words using PostgreSQL's split-like function called **string_to_array()**. And then we can use the PostgresSQL **unnest()** function to turn the resulting array into separate rows.

```
pg4e=> string_to_array('Hello world', ' ');
 string_to_array
-----------------
 {Hello,world}

pg4e=> unnest(string_to_array('Hello world', ' '));
 unnest
--------
 Hello
 world
```

After that, it is just a few **SELECT DISTINCT** statements and we can create and use an inverted index.

```
CREATE TABLE docs (id SERIAL, doc TEXT, PRIMARY KEY(id));
INSERT INTO docs (doc) VALUES
('This is SQL and Python and other fun teaching stuff'),
('More people should learn SQL from UMSI'),
('UMSI also teaches Python and also SQL');

CREATE TABLE docs_gin (
  keyword TEXT,
  doc_id INTEGER REFERENCES docs(id) ON DELETE CASCADE
);

...

pg4e=> select * from docs_gin;
 keyword  | doc_id
----------+--------
 Python   |      1
 SQL      |      1
 This     |      1
 stuff    |      1
 teaching |      1
 More     |      2
 SQL      |      2
 UMSI     |      2
 from     |      2
 learn    |      2
 people   |      2
 should   |      2
 Python   |      3
 SQL      |      3
 UMSI     |      3
 also     |      3
```

```
 and      |      3
 teaches  |      3
(22 rows)
```

References

- Split column into multiple rows in Postgres (Stackoverflow)

Inverted Indexes in PostgreSQL ⏪📖🏠⏩

- Generalized Inverse Index (GIN)
- Generalized Search Tree (GiST)

GIN indexes are the preferred text search index type. Advantages: exact matches, efficient on lookup/search. Disadvantages: can be costly when inserting or updating data because every new word is inserted somewhere in the index and can get large. Like the B-Tree, the GIN is the usual "go-to" inverted index and GiST is used in more special cases. The previous example was a rough approximation of a GIN index.

Hashing is used to reduce the size of and cost to update the GiST. *A GiST index is lossy, meaning that the index might produce false matches, and it is necessary to check the actual table row to eliminate such false matches. (PostgreSQL does this automatically when needed.)* The indexes have equivalent functionality and will return the same rows - but there may be performance / storage tradeoffs between GIN and GiST.

Both GIN and GiST want to know something about the type of array data it will be indexing and the kinds of operations that we will be using in **WHERE** clauses. In the example below we are indexing arrays of strings (i.e. text[]) and will be using the "<@" operator (contained within).

We can build a simple GIN index like the manual index above:

```
CREATE TABLE docs (id SERIAL, doc TEXT, PRIMARY KEY(id));

CREATE INDEX gin1 ON docs USING gin(string_to_array(doc, ' ')  array_ops);

INSERT INTO docs (doc) VALUES
('This is SQL and Python and other fun teaching stuff'),
('More people should learn SQL from UMSI'),
('UMSI also teaches Python and also SQL');
```

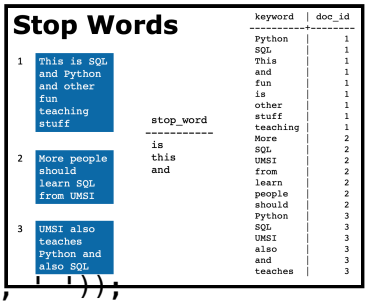
The <@ is looking for an intersection between two arrays

```
SELECT id, doc FROM docs WHERE '{learn}' <@ string_to_array(doc, ' ');

 id |                  doc
----+---------------------------------------
  2 | More people should learn SQL from UMSI

EXPLAIN SELECT id, doc FROM docs WHERE '{learn}' <@ string_to_array(doc, ' ');

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on docs  (cost=12.05..21.53 rows=6 width=32)
   Recheck Cond: ('{learn}'::text[] <@ string_to_array(doc, ' '::text))
   ->  Bitmap Index Scan on gin1  (cost=0.00..12.05 rows=6 width=0)
         Index Cond: ('{learn}'::text[] <@ string_to_array(doc, ' '::text))
```

References

- GIN and Gist Indexes in PostgreSQL

Inverted Indexes of Natural Language - Stemming and Stop Words ⏪ 📖 🏠 ⏩



To take advantage of the "naruralness" of natural language, we need to ignore words that convey no meaning and consistently reduce variations of words with equivalent meanings down to a single "stem word".

Recall this failure

```
SELECT DISTINCT id, doc FROM docs AS D
JOIN docs_gin AS G ON D.id = G.doc_id
WHERE G.keyword = ANY(string_to_array('Search for Lemons and Neons', ' '));

 id |                          doc
----+-------------------------------------------------------
  1 | This is SQL and Python and other fun teaching stuff
  3 | UMSI also teaches Python and also SQL
```

The word "and" contributed no real meaning to our query. And it took up valuable space in our GIN index. So we put it on the stop word list. Lets implement stop word and stemming capabilities by hand and then just use PostgreSQL features to build a natural language search.

```
SELECT * FROM stop_words;
 word
------
 is
 this
 and

SELECT * FROM docs_stem;
   word    | stem
-----------+-------
 teaching | teach
 teaches  | teach

...

SELECT * FROM docs_gin;

 keyword | doc_id
---------+--------
 also    |      3
 from    |      2
 fun     |      1
 learn   |      2
 more    |      2
 other   |      1
 people  |      2
 python  |      1
 python  |      3
 should  |      2
 sql     |      3
 sql     |      1
 sql     |      2
 stuff   |      1
 teach   |      3
 teach   |      1
 this    |      1
 umsi    |      2
 umsi    |      3
(19 rows)
```

Stemming and stop words (and the meaning of "meaning") depend on which language is stored in the text column. The default install of PostgreSQL knows the rules for a few languages and more can be installed:

```
SELECT cfgname FROM pg_ts_config;

  cfgname
------------
 simple
 danish
 dutch
 english
 finnish
 french
 german
 hungarian
 italian
 norwegian
 portuguese
 romanian
 russian
 spanish
 swedish
 turkish
(16 rows)
```

Text Search Functions ⏪ 📖 🏠 ⏩

PostgreSQL provides some functions that turn a text document/string into an "array" with stemming, stop words, and other language-oriented features.

ts_vector() returns a list of words that represent the document. **ts_query()** returns a list of words with operators to representaions various logical combinations of words much like Google's Advanced Search.

```
SELECT to_tsvector('english', 'UMSI also teaches Python and also SQL');

                   to_tsvector
-------------------------------------------------
 'also':2,6 'python':4 'sql':7 'teach':3 'umsi':1

SELECT to_tsquery('english', 'teaching');

 to_tsquery
------------
 'teach'
```

In a **WHERE** clause we use the @@ operator to ask is a **ts_query** matches a **ts_vector**.

```
SELECT to_tsquery('english', 'teaching') @@
  to_tsvector('english', 'UMSI also teaches Python and also SQL');

 ?column?
----------
 t
```

References

- Introduction to Text Search (PostgreSQL)
- Operators and functions that work with ts_query and ts_vector (PostgreSQL)

Making a Natural Language Inverted Index with PostgreSQL

As you might expect, letting PostgreSQL do all the work is the easy part. Stop words and stems are all handled in the "ts_" functions. And the GIN knows what operations you will be using automatically when you pass in a **ts_vector**.

```
CREATE TABLE docs (id SERIAL, doc TEXT, PRIMARY KEY(id));

CREATE INDEX gin1 ON docs USING gin(to_tsvector('english', doc));

INSERT INTO docs (doc) VALUES
('This is SQL and Python and other fun teaching stuff'),
('More people should learn SQL from UMSI'),
('UMSI also teaches Python and also SQL');

SELECT id, doc FROM docs WHERE
    to_tsquery('english', 'learn') @@ to_tsvector('english', doc);

 id |                  doc
----+---------------------------------------
  2 | More people should learn SQL from UMSI

EXPLAIN SELECT id, doc FROM docs WHERE
    to_tsquery('english', 'learn') @@ to_tsvector('english', doc);

                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on docs  (cost=12.05..23.02 rows=6 width=36)
   Recheck Cond: ('''learn'''::tsquery @@ to_tsvector('english'::regconfig, doc))
   ->  Bitmap Index Scan on gin1  (cost=0.00..12.05 rows=6 width=0)
         Index Cond: ('''learn'''::tsquery @@ to_tsvector('english'::regconfig, doc))
```

References

- Controlling Text Search (PostgreSQL)
- FULLTEXT query with scores/ranks in PostgreSQL (Stackoverflow)
- Best way to use PostgreSQL full text search ranking (Stackoverflow)

There is a lot of ways to index and search in PostgreSQL

You can ask PostgreSQL the different index / **WHERE** clause operator combinations it supports. There are quite a few and they can change from one PostgreSQL version to another.

```
SELECT am.amname AS index_method, opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;

 index_method |      opclass_name
--------------+------------------------
 brin         | abstime_minmax_ops
 brin         | bit_minmax_ops
 brin         | box_inclusion_ops
 ...
 brin         | uuid_minmax_ops
 brin         | varbit_minmax_ops
 btree        | abstime_ops
 btree        | array_ops
 ...
 btree        | varbit_ops
```

```
  btree         | varchar_ops
  btree         | varchar_pattern_ops
  gin           | array_ops
  gin           | jsonb_ops
  gin           | jsonb_path_ops
  gin           | tsvector_ops
  gist          | box_ops
  gist          | circle_ops
  ...
  gist          | tsvector_ops
  hash          | abstime_ops
  hash          | aclitem_ops
  hash          | array_ops
  ...
  hash          | varchar_pattern_ops
  hash          | xid_ops
  spgist        | box_ops
  spgist        | inet_ops
  spgist        | kd_point_ops
  spgist        | poly_ops
  spgist        | quad_point_ops
  spgist        | range_ops
(134 rows)
```

Copyright Charles R. Severance, CC0 - You are welcome to adapt, reuse or reference this material with or without attribution.

Feel free to help improve this lecture at GitHub.