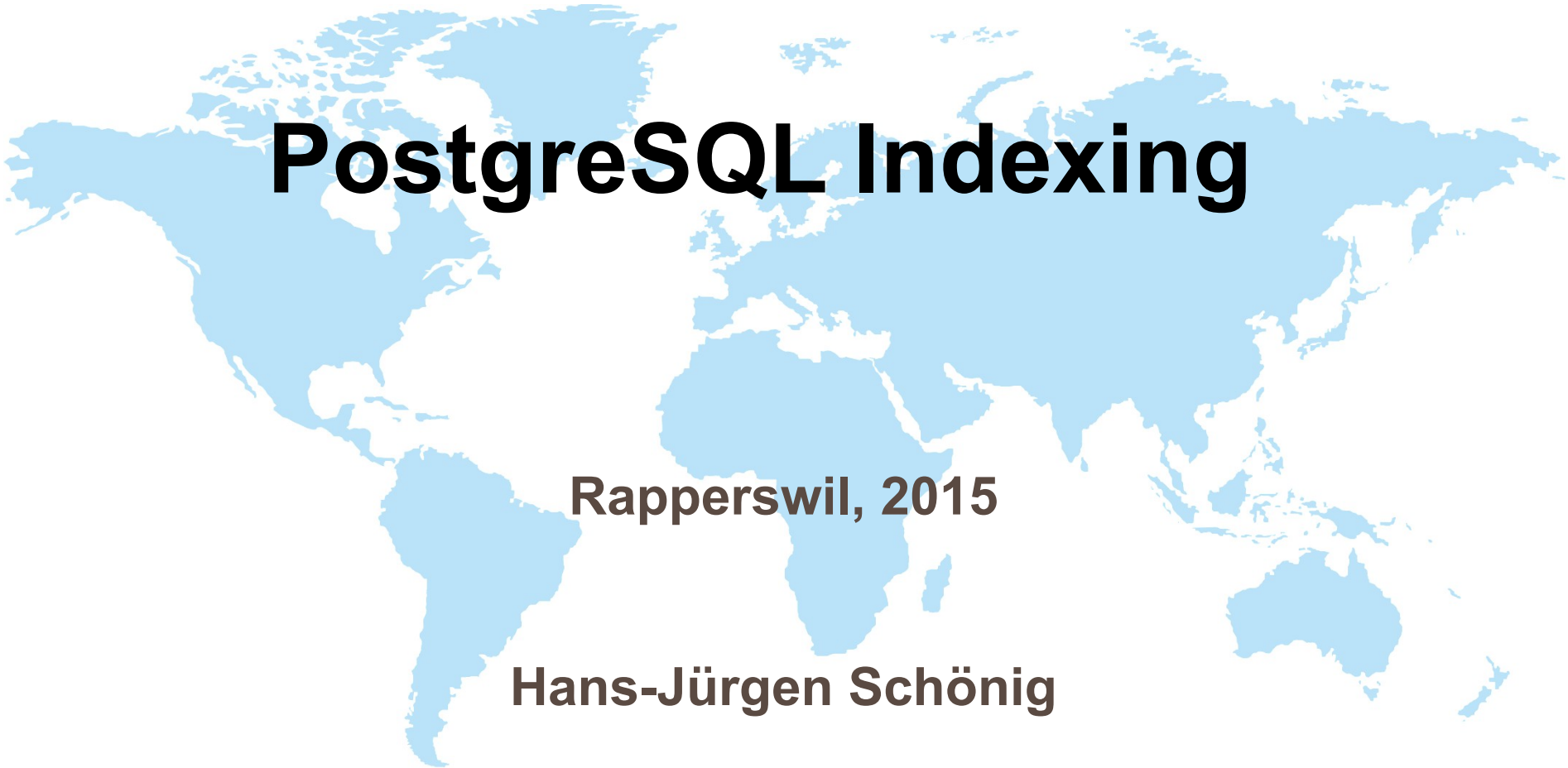


PostgreSQL Indexing

Rapperswil, 2015

Hans-Jürgen Schönig

Scope of this session:

- What a basic index does
- **The PostgreSQL optimizer (cost model)**
- Classical B-tree Indexes
- Partial / functional indexes
- Different types of indexes
- Full-Text-Search
- Fuzzy matching
- Writing your own indexing strategy

1. Basic indexing:

- Generating test data:

  - for the purpose of this session we need a table consisting of two columns:

```
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name) VALUES ('hans');
INSERT 0 1
test=# INSERT INTO t_test (name) VALUES ('paul');
INSERT 0 1
```

1. Basic indexing:

- A lot more test data ...

  - Let us create some more test data
    by repeating the process

```
test=# INSERT INTO t_test (name) SELECT name FROM t_test;
INSERT 0 2

...

test=# INSERT INTO t_test (name) SELECT name FROM t_test;
INSERT 0 2097152
```

1. Basic indexing:

- Reading some data:

  - Let us see, how PostgreSQL executes a simple query:

```
test=# SELECT count(*) FROM t_test;
  count
 ---------
 4194304
(1 row)

Time: 431.192 ms


test=# explain analyze SELECT count(*) FROM t_test;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=75100.80..75100.81 rows=1 width=0) (actual time=977.865..977.865 rows=1 loops=1)
   ->  Seq Scan on t_test  (cost=0.00..64615.04 rows=4194304 width=0)
                          (actual time=0.013..531.448 rows=4194304 loops=1)
 Total runtime: 977.917 ms
(3 rows)

Time: 1045.065 ms
```

1. Basic indexing:

- Reading some data:

  - Let us add a filter:

```
test=# SELECT count(*) FROM t_test WHERE id = 421234;
 count
-------
     1
(1 row)

Time: 476.965 ms

test=# explain analyze SELECT count(*) FROM t_test WHERE id = 421234;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=75100.80..75100.81 rows=1 width=0) (actual time=495.134..495.135 rows=1 loops=1)
           ->  Seq Scan on t_test  (cost=0.00..75100.80 rows=1 width=0)
                     (actual time=53.405..495.126 rows=1 loops=1)
           Filter: (id = 421234)
           Rows Removed by Filter: 4194303
 Total runtime: 495.175 ms
(5 rows)

Time: 520.659 ms
```

1. Basic indexing:

- Sequentially reading data:

  - In case you like reading the phone book sequentially
    we are basically done.

  - Sequentially reading the phone book is technically ok

    => but socially not accepted

  - Defining an index is the desired solution

1. Basic indexing:

- Creating an index

```
test=# \h CREATE INDEX
Command:     CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ]
        ON table_name [ USING method ]
( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ]
        [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace_name ]
[ WHERE predicate ]
```

- At the end of the day all clauses will be covered by this training

1. Basic indexing:

- A typical index:

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
Time: 7357.663 ms
```

- This gives us a standard btree index

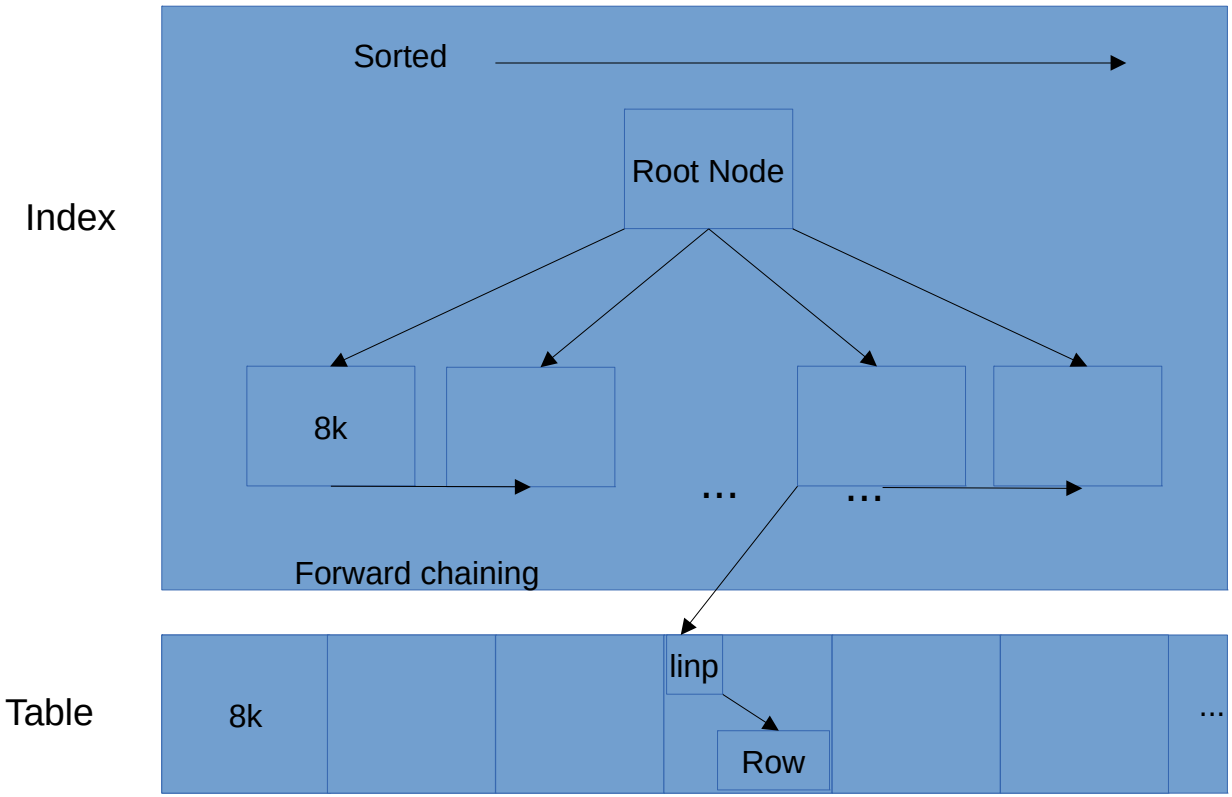
- PostgreSQL provides “High-Concurrency B-Trees”
  (Lehman-Yao, 1981)

- Many people can modify the index at the same time

- Highly efficient B+ tree

1. Basic indexing:

- How a btree works:



1. Basic indexing:

- Indexing is beneficial

```
test=# explain analyze SELECT count(*)
                    FROM        t_test
                    WHERE       id = 421234;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=8.73..8.74 rows=1 width=0)
                    (actual time=0.024..0.024 rows=1 loops=1)
          ->  Index Only Scan using idx_id on t_test  (cost=0.00..8.73 rows=1 width=0)
                    (actual time=0.019..0.020 rows=1 loops=1)
          Index Cond: (id = 421234)
                    Heap Fetches: 1
Total runtime: 0.057 ms
(5 rows)

Time: 0.395 ms
```

- A lot faster :).

1. Basic indexing:

- Still slow ...

```
test=# SELECT count(*) FROM t_test WHERE name = 'hans';
  count
---------
2097152
(1 row)

Time: 787.407 ms
```

- This is still slow. Let us create an index ...

```
test=# CREATE INDEX idx_name ON t_test (name);
CREATE INDEX
```

1. Basic indexing:

- The benefit is exactly zero:

```
test=# SELECT count(*) FROM t_test WHERE name = 'hans';
  count
---------
 2097152
(1 row)

Time: 782.443 ms
test=# explain SELECT count(*) FROM t_test WHERE name = 'hans';
                              QUERY PLAN
-----------------------------------------------------------------------
Aggregate  (cost=80350.32..80350.33 rows=1 width=0)
   ->  Seq Scan on t_test  (cost=0.00..75100.80 rows=2099808 width=0)
Filter: (name = 'hans'::text)
(3 rows)
```

- The index won't be used

- Too many identical values (“not selective”)

1. Basic indexing:

- The cost is far from zero:

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
----------------
 177 MB
(1 row)

test=# SELECT pg_size_pretty(pg_relation_size('idx_id'));
 pg_size_pretty
----------------
 90 MB
(1 row)

test=# SELECT pg_size_pretty(pg_relation_size('idx_name'));
 pg_size_pretty
----------------
 90 MB
(1 row)
```

- Indexes need a fair amount of space

1. Basic indexing:

- Input values DO make a difference:

```
test=# explain SELECT count(*) FROM t_test WHERE name = 'hans';
                        QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=80350.32..80350.33 rows=1 width=0)
   ->  Seq Scan on t_test  (cost=0.00..75100.80 rows=2099808 width=0)
         Filter: (name = 'hans'::text)
(3 rows)

test=# explain SELECT count(*) FROM t_test WHERE name = 'hans2';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=7.74..7.75 rows=1 width=0)
   ->  Index Only Scan using idx_name on t_test  (cost=0.00..7.74 rows=1 width=0)
         Index Cond: (name = 'hans2'::text)
(3 rows)
```

- PostgreSQL will decide depending on the input value

=> cost based optimization

1. Basic indexing:

- Partial indexes:

  - In our example the index is only used in case of rare or non-existing values

  - What is the point of an index when its entire content is totally useless?

  => a more selective strategy is needed

1. Basic indexing:

- Partial indexes:

```
test=# DROP INDEX idx_name;
DROP INDEX

test=# CREATE INDEX idx_name ON t_test (name)
          WHERE name NOT IN ('hans', 'paul');
CREATE INDEX

test=# SELECT pg_size_pretty(pg_relation_size('idx_name'));
 pg_size_pretty
----------------
 8192 bytes
(1 row)
```

- A partial index reduces space consumption

- Benefit is still the same

1. Basic indexing:

- Equal benefit – lower cost:

```
test=# explain SELECT count(*) FROM t_test WHERE name = 'hans';
                        QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=80350.32..80350.33 rows=1 width=0)
   ->  Seq Scan on t_test  (cost=0.00..75100.80 rows=2099808 width=0)
         Filter: (name = 'hans'::text)
(3 rows)

test=# explain SELECT count(*) FROM t_test WHERE name = 'hans2';
                                 QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=7.28..7.29 rows=1 width=0)
   ->  Index Only Scan using idx_name on t_test  (cost=0.00..7.28 rows=1 width=0)
         Index Cond: (name = 'hans2'::text)
(3 rows)
```

- This is exactly the same as before !

1. Basic indexing:

- What about functions?

```
test=# CREATE INDEX idx_cos ON t_test ( cos(id) );
CREATE INDEX
Time: 16867.228 ms

test=# explain SELECT count(*) FROM t_test WHERE cos(id) = 17;
                                QUERY PLAN
---------------------------------------------------------------------------------
 Aggregate  (cost=23960.99..23961.00 rows=1 width=0)
   ->  Bitmap Heap Scan on t_test  (cost=395.25..23908.56 rows=20972 width=0)
         Recheck Cond: (cos((id)::double precision) = 17::double precision)
         ->  Bitmap Index Scan on idx_cos  (cost=0.00..390.01 rows=20972 width=0)
               Index Cond: (cos((id)::double precision) = 17::double precision)
(5 rows)
```

- PostgreSQL provides functional indexes

- VERY nice to avoid additional columns

- Gives a lot of extra flexibility

- The output of the function is stored inside the tree; not the expression
  => we don't calculate things during a scan

1. Basic indexing:

- Type of functions allowed

- Functions must be deterministic

=> “immutable”

=> Functions can be written in almost any language

=> This is highly performance sensitive

2. The PostgreSQL cost model

- How does PostgreSQL decide on index vs. no index?

  - PostgreSQL uses statistics to estimate the number of rows coming back

  - Each operation will be assigned to costs

    => costs are just a number to compare different options inside the planner

  - Costs parameters can be changed at runtime or globally

    => be careful, it can go against you

2. The PostgreSQL cost model

- pg_stats is your friend:

```
test=# \d pg_stats
          View "pg_catalog.pg_stats"
        Column          |   Type    | Modifiers
------------------------+-----------+-----------
 schemaname             | name      |
 tablename              | name      |
 attname                | name      |
 inherited              | boolean   |
 null_frac              | real      |
 avg_width              | integer   |
 n_distinct             | real      |
 most_common_vals       | anyarray  |
 most_common_freqs      | real[]    |
 histogram_bounds       | anyarray  |
 correlation            | real      |
 most_common_elems      | anyarray  |
 most_common_elem_freqs | real[]    |
 elem_count_histogram   | real[]    |
```

2. The PostgreSQL cost model

- Updating statistics

  - System statistics are updated by ANALYZE:

```
test=# \h ANALYZE
Command:     ANALYZE
Description: collect statistics about a database
Syntax:
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

- In most setups autovacuum is in charge
  of updating pg_statistic

- In most cases statistics are not an issue

2. The PostgreSQL cost model

- How does PostgreSQL estimate costs?

  - seq_page_cost = 1
  - random_page_cost = 4
  - cpu_tuple_cost = 0.01
  - cpu_operator_cost = 0.0025
  - cpu_index_tuple_cost = 0.005

2. The PostgreSQL cost model

- Let us do the math (1):

```
test=# explain SELECT count(*) FROM t_test;
                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=75100.80..75100.81 rows=1 width=0)
   ->  Seq Scan on t_test  (cost=0.00..64615.04 rows=4194304 width=0)
(2 rows)
```

- total costs are at 75100.81

- costs are composed of I/O and CPU costs

- there are “total costs” and “startup costs”
  => what does it take to start an operation?
  Maybe a node requires sorted input or some other preprocessing

2. The PostgreSQL cost model

- Let us do the math (2):

```
test=# SELECT pg_relation_size('t_test') / 8192;
 ?column?
----------
    22672
(1 row)
```

- our table consists of 22672 blocks

- each block is 8kb in size

2. The PostgreSQL cost model

- Let us do the math (3):

The seq scan:

I/O cost = 22672 * seq_page_cost = 22672
4.194.304 * cpu_tuple_cost = 41943.04

= 64615.04 for the seq scan

The aggregate:

4.194.304 * cpu_operator_cost = 10485.76

Total costs => 75.100.80 + cpu_operator_cost
(we have to display the tuple)

2. The PostgreSQL cost model

- Inflation at work:

```
test=# SET seq_page_cost TO 10;
SET

test=# explain SELECT count(*) FROM t_test;
                            QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=279148.80..279148.81 rows=1 width=0)
   ->  Seq Scan on t_test  (cost=0.00..268663.04 rows=4194304 width=0)
(2 rows)
```

- Costs can be changed at runtime to fine tune index usage

=> only do this if you are fully aware of what you are doing. It can have unintended side effects

2. The PostgreSQL cost model

- Spinning disks vs. SSDs

- Traditional disks are fast sequentially
  and pretty bad when doing random I/O

- SSDs fixed the problem.

=> consider changing random_page_cost

2. The PostgreSQL cost model

- Abusing tablespaces:

```
test=# ALTER TABLESPACE pg_default
        SET (random_page_cost = 1);
ALTER TABLESPACE
```

- Allows different cost settings for various disk subsystems

- It also allows to split “cached” and “uncached” data -> ugly but useful

2. The PostgreSQL cost model

- Correlation and disk layout

```
test=# CREATE TABLE t_random AS SELECT *
        FROM        t_test
        ORDER BY random();
SELECT 4194304
test=# CREATE INDEX idx_random ON t_random(id);
CREATE INDEX
test=# ANALYZE t_random;
ANALYZE
```

- The PostgreSQL optimizer considers the physical order of rows on disk

- High-correlation will make indexes ways more likely as the optimizer reduces its estimates for I/O costs.

2. The PostgreSQL cost model

- Correlation and disk layout

```
test=# explain SELECT count(*) FROM t_test WHERE id < 1000;
                                QUERY PLAN
-------------------------------------------------------------------------
Aggregate  (cost=75.35..75.36 rows=1 width=0)
  ->  Index Only Scan using idx_id on t_test
                (cost=0.00..72.72 rows=1049 width=0)
        Index Cond: (id < 1000)
(3 rows)
```

```
test=# explain SELECT count(*) FROM t_random WHERE id < 1000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=950.31..950.32 rows=1 width=0)
  ->  Index Only Scan using idx_random on t_random
                (cost=0.00..947.94 rows=947 width=0)
        Index Cond: (id < 1000)
(3 rows)
```

2. The PostgreSQL cost model

- Implications:

  - This is why different plans can pop up
    EVEN if the data is the same

  - There is no fixed amount of data making
    PostgreSQL switch from index to
    sequential scan

  - High correlation can improve performance

    => consider clustering the table

```
test=# \h CLUSTER
Command:     CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

3. Indexing many columns

- Using OR / AND:

  - PostgreSQL can use more than one index per table per query

  - PostgreSQL provides multi-column indexes

  - What you might see is a so called “Bitmap Scan”

    => don't mix it up with Oracle Bitmap Indexes

3. Indexing many columns

- Bitmap scans:

```
test=# explain SELECT * FROM t_test WHERE id = 2343 OR id = 423423;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on t_test  (cost=9.44..17.41 rows=2 width=9)
   Recheck Cond: ((id = 2343) OR (id = 423423))
   ->  BitmapOr  (cost=9.44..9.44 rows=2 width=0)
         ->  Bitmap Index Scan on idx_id  (cost=0.00..4.72 rows=1 width=0)
               Index Cond: (id = 2343)
         ->  Bitmap Index Scan on idx_id  (cost=0.00..4.72 rows=1 width=0)
               Index Cond: (id = 423423)
(7 rows)
```

- PostgreSQL will scan the index twice

- PostgreSQL will look for blocks in the underlying table

- The condition has to be re-evaluated

  The TID bitmap has a concept of lossy page: if there seem to be too many matching tuples on heap page, the page is remembered by the bitmap as whole, instead of individual tuples. That saves space in the bitmap, but also requires the recheck of each tuple.

  In other cases the reason for recheck is that the index is not able to to decide on matching tuples, e.g. when GIN is used to evaluate <@ operator for an array.

3. Indexing many columns

- Bitmap scans:

```
test=# explain SELECT * FROM t_test WHERE id = 2343 AND name = 'josef';
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using idx_name on t_test  (cost=0.00..8.27 rows=1 width=9)
   Index Cond: (name = 'josef'::text)
   Filter: (id = 2343)
(3 rows)
```

- PostgreSQL does not always use two indexes
  when you have 2 quals

- The more selective index might be enough

3. Indexing many columns

- Multicolumn indexes:

```
test=# DROP INDEX idx_id;
DROP INDEX

test=# CREATE INDEX idx_combined ON t_test (id, name);
CREATE INDEX
test=# explain SELECT * FROM t_test WHERE id = 10;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Index Only Scan using idx_combined on t_test  (cost=0.00..8.91 rows=1 width=9)
   Index Cond: (id = 10)
(2 rows)
```

- PostgreSQL can use parts of those column IF they are in the first part(s) of the index

- Imagine a phone book; it is just liked a combined index

3. Indexing many columns

- Many indexes or combined indexes?
  - It depends on what you want to query
  - If you always use the first conditions in the index
    a combined index might be a good idea
  - Many indexes are more flexible but maybe not perfect
  - Sometimes a mixed-strategy can be useful

4. Indexes to provide order

- b-tress can be used for more than searching

  - Binary trees provide you with order.

  - Order helps to avoid repeated sorting.

```
test=# explain SELECT * FROM t_test ORDER BY id LIMIT 10;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..0.31 rows=10 width=9)
   ->  Index Scan using idx_id on t_test  (cost=0.00..131602.27 rows=4194304 width=9)
(2 rows)
```

5. Dealing with upper / lowercase

- Upper and lower case searches are common:
  - If you want to do case-insensitive, don't use a functional index
  - Consider using “citext”

```
test=# CREATE EXTENSION citext;
CREATE EXTENSION
test=# SELECT 'ABC'::citext = 'abc'::citext;
 ?column?
----------
 t
(1 row)
```

6. Different types of indexes

- PostgreSQL supports more than just btrees

  - B-Trees are fine if you are interested in things which can be sorted

  - Try to sort polygons => you won't find them

  - Geometric data and Full-Text-Search need different algorithms

NOTE: This is not about, which index is faster.
This is about the correct ALGORITHM

6. Different types of indexes

- Index types provided by PostgreSQL
  - B-Trees
  - Gist: Generalized Search Tree
  - Gin: Generalized Inverted Index
  - Sp-Gist: Space Partitioned Gist
  - Hash
  - Brin indexes (from 9.5 on)

7. Gist indexes

- Gist operates on different principles
  than btree

  - it supports “contains”, “left of”, “overlaps”, etc.

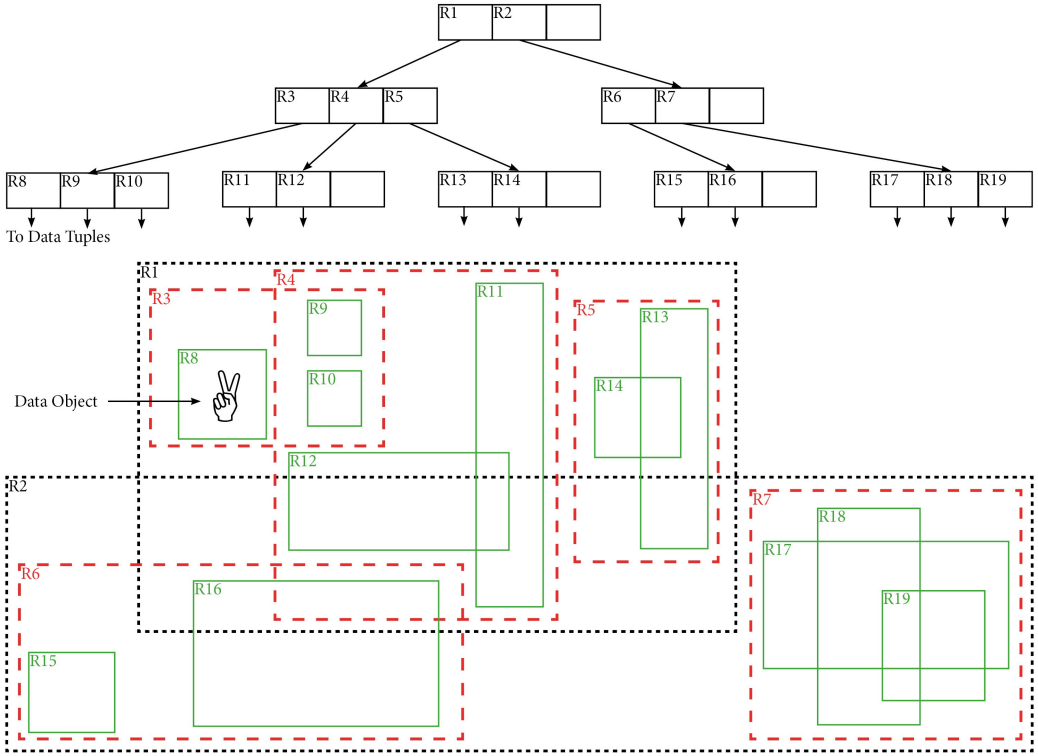
  - “contains”, etc. are good for

    => Full Text Search (better use GIN)
    => Geometric operations (PostGIS, etc.)
    => Finding genome sequences
    => Handling ranges (time, etc.)
    => Fuzzy search

  - Gist allows KNN-search

7. Gist indexes

- How it works internally ...



7. GIN indexes

- GIN is a so called inverted index

  - Used for Full Text Search

  - If you have 1 mio documents containing the word "house". Do you really want to have house inside the index 1 mio times?

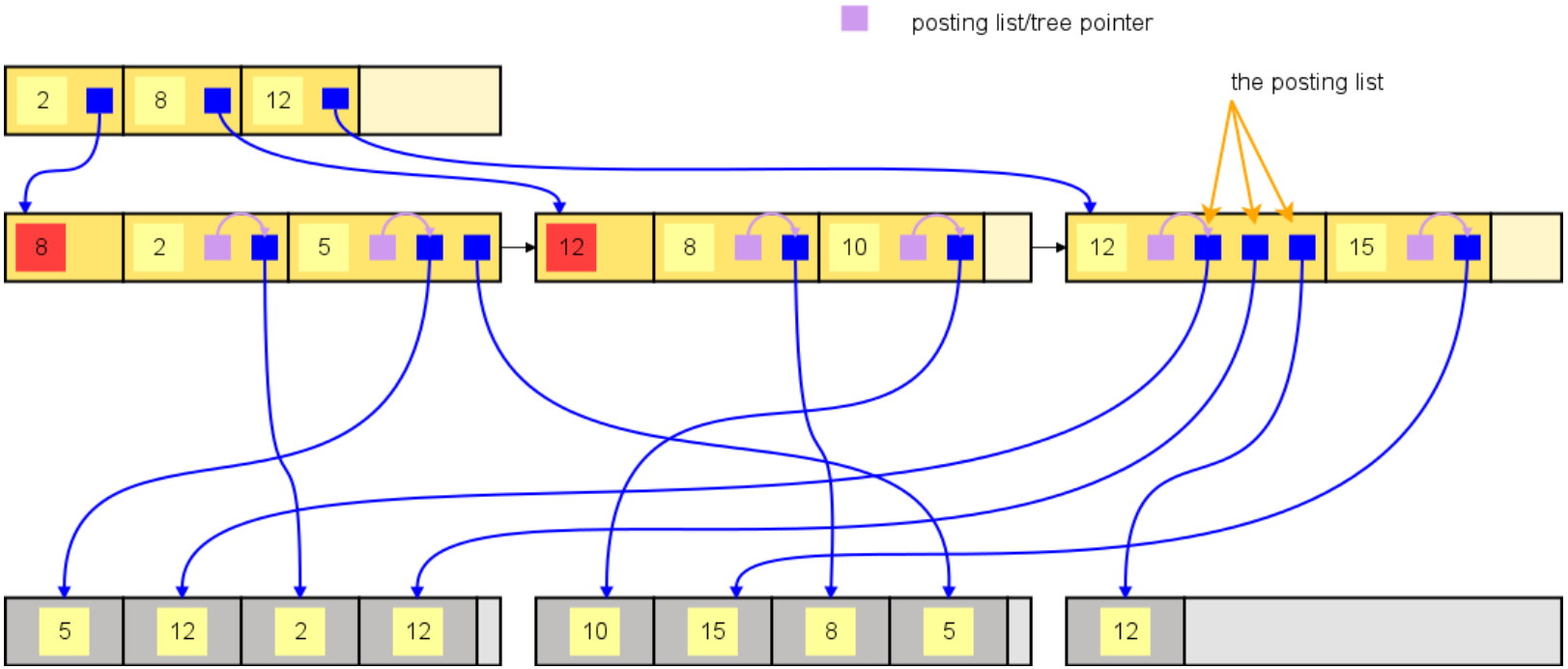
    => Binary tree for words
    => A document list for each word

    => Classical approach to text search

  - FTS is not about "=", it is about "contains"
    => forget btree

7. GIN indexes

- GIN internal workings:



7. SP-Gist indexes

- SP-Gist is a space partitioned index

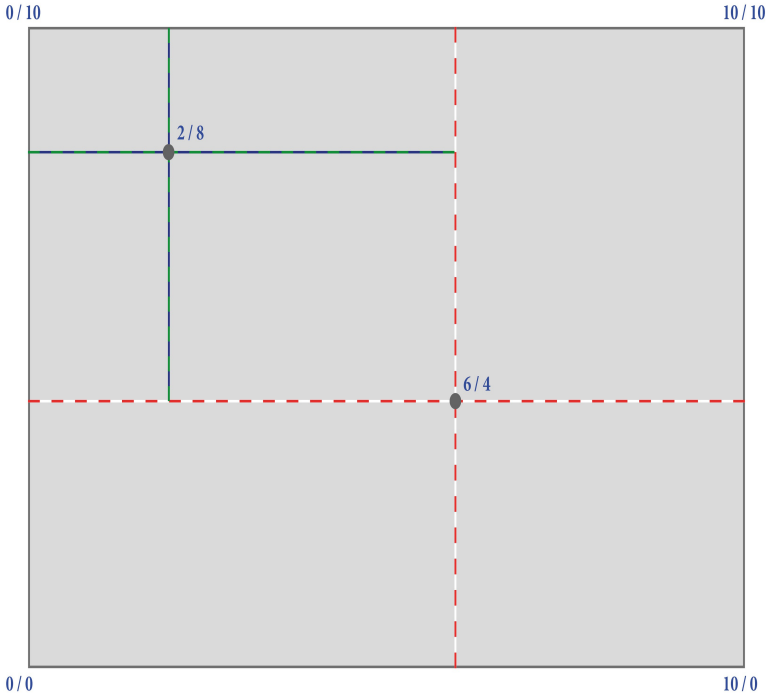
  - Can be used for a variety of algorithms, which use space partitioning

    => quad trees
    => suffix trees
    => k-d trees

7. SP-Gist indexes

- Quad trees: A prototype example ...

  - We want to insert ... (6, 4) and (2, 8)



8. Full Text Search

- Stemming:
  - Before searching, it makes sense to perform "stemming"

```
test=# SELECT to_tsvector('english', 'having many cars is better than
                    to have just one car');
      to_tsvector
----------------------------------------
 'better':5 'car':3,11 'mani':2 'one':10
(1 row)
```

8. Full Text Search

- Stemming is language dependent:

  - Stemming works nicely for “roman” languages
    => it is hard to do this for chinese and so on

```
test=# SELECT       to_tsvector('english', 'i am'),
                    to_tsvector('german', 'i am'),
                    to_tsvector('dutch', 'i am');
          to_tsvector | to_tsvector | to_tsvector
-------------+-------------+--------------
                      | 'i':1       | 'am':2 'i':1
(1 row)
```

8. Full Text Search

- “contains” is your friend:
  - ts_query compares a search string with a so called ts_vector:

```
test=# SELECT to_tsvector('english', 'having many cars is better
                                    than to have just one car')
                    @@ to_tsquery('english', 'car');
 ?column?
----------
 t
(1 row)
```

8. Full Text Search

- Indexing is easy:
  - All you need is a functional index
  - Alternatively the stemmed content can be “materialized” in a separate column

```
CREATE INDEX idx_fti ON t_test
        USING gist (to_tsvector('german', name));
```

8. Full Text Search

- ts_vector and ts_query magic

  - PostgreSQL allows you to use “and” (&) and “or” (|)

```
test=# SELECT to_tsvector('english', 'having many cars is better than
                                    to have just one car')
                          @@ to_tsquery('english', 'car & truck');
 ?column?
----------
 f
(1 row)

test=# SELECT to_tsvector('english', 'having many cars is better than
                                    to have just one car')
                          @@ to_tsquery('english', '(car | truck) & many');
 ?column?
----------
 t
(1 row)
```

8. Full Text Search

- A stupid question: What is a “word”?

  - PostgreSQL is NOT limited to textual search

  - Remember, it is all about “contains” ...

  - Create yourself your own parser:

```
test=# \h CREATE TEXT SEARCH PARSER
Command:     CREATE TEXT SEARCH PARSER
Description: define a new text search parser
Syntax:
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

8. Full Text Search

- Even more flexibility:

```
test=# \h CREATE TEXT SEARCH DICTIONARY
Command:     CREATE TEXT SEARCH DICTIONARY
Description: define a new text search dictionary
Syntax:
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)

test=# \h CREATE TEXT SEARCH TEMPLATE
Command:     CREATE TEXT SEARCH TEMPLATE
Description: define a new text search template
Syntax:
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

8. Full Text Search

- Even more flexibility (2):

```
test=# \h CREATE TEXT SEARCH CONFIGURATION
Command:     CREATE TEXT SEARCH CONFIGURATION
Description: define a new text search configuration
Syntax:
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

9. Operator classes

- What does it take to organize a btree?

Operator	Strategy number
<	1
<=	2
=	3
>=	4
<	5

9. Operator classes

- Why care?

  - The way numbers are treated is pretty “common”

  - How about sorting this one?

    “2305 09 04 78”
    “4353 07 06 77”

    => it seems the sort order is correct as shown
    => it isn't – it is an Austrian social security number

    => 1977 was before 1978 and not other way round

9. Operator classes

- Defining indexing strategies

  - We can write our own operators

  - Those operators can be assigned to an operator class, which will tell the index how to “behave”

    “2305 09 04 78”
    “4353 07 06 77”

    => it seems the sort order is correct as shown
    => it isn't – it is an Austrian social security number

    => 1977 was before 1978 and not other way round

9. Operator classes

- Writing an operator (1):

```
test=# CREATE OR REPLACE FUNCTION normalize_si(text)
        RETURNS text AS $$
BEGIN
        RETURN substring($1, 9, 2) ||
                substring($1, 7, 2) ||
                substring($1, 5, 2) ||
                substring($1, 1, 4);
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
CREATE FUNCTION

test=# SELECT normalize_si('2305090478');
 normalize_si
--------------
 7804092305
(1 row)
```

9. Operator classes

- Writing an operator (2):

```
test=# CREATE OR REPLACE FUNCTION si_lt(text, text)
        RETURNS boolean AS
$$
        BEGIN
                RETURN normalize_si($1) < normalize_si($2);
        END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;

test=# CREATE OPERATOR <# (
                PROCEDURE=si_lt,
                LEFTARG=text,
                RIGHTARG=text);
CREATE OPERATOR

CREATE FUNCTION
test=# SELECT '2305090478'::text <# '4353070677'::text;
 ?column?
----------
 f
(1 row)
```

The operator function will only be used for sequential scan, but the "index support functions" are actually the workhorse of the index scan. (The "operator functions" are only used during the index scan if recheck is involved.) Also, the support functions look "the same" only for B-tree.

9. Operator classes

- Creating the operator class:

- write operators for all operations needed
- write “support functions” (= “same”, etc.)
- make sure that the most important strategies
  have proper operators

```
test=# \h CREATE OPERATOR CLASS
Command:     CREATE OPERATOR CLASS
Description: define a new operator class
Syntax:
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ]
                  [ FOR SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
                           function_name ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

10. Available operator classes

- pg_trgm

  - Trigrams are perfect to perform fuzzy matching

  - Trigrams can be used nicely along with KNN-search

  - pg_trgm is available as extension to PostgreSQL

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

  - Problem: “What is the proper way to spell the name of this village?

    “gramatneusiedl” vs. “grammatneusiedel”?

10. Available operator classes

- Testing pg_trgm

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE

test=# COPY t_location FROM
       PROGRAM 'curl www.cybertec.at/secret/orte.txt';
COPY 2354

test=# CREATE INDEX idx_trgm
            ON t_location
            USING gist (name gist_trgm_ops);
CREATE INDEX
```

10. Available operator classes

- Testing pg_trgm (2):

```
test=# SELECT *
        FROM       t_location
        ORDER BY name <-> 'kramertneusiedel'
        LIMIT 5;
      name
----------------
 Gramatneusiedl
 Klein-Neusiedl
 Potzneusiedl
 Kramsach
 Neusiedl am See
(5 rows)
```

11. Traditional LIKE

- LIKE can be indexed (btree) in some cases:

- The PostgreSQL optimizer can rewrite queries featuring LIKE
  in a fancy and efficient way

  => The goal is to find the “next character” in line
  and query for a range

- This kind of rewrite only works when the next character
  It actually knows to PostgreSQL

- Special operator classes might be needed

  => varchar_pattern_ops, text_pattern_ops

11. Indexing regular expressions

- Regular expressions can be represented as graphs

```
test=# SELECT * FROM t_location WHERE name ~ 'P[uo].*sied(e)?l';
          name
--------------------------
 Purbach am Neusiedler See
 Potzneusiedl
(2 rows)
```

```
test=# explain SELECT * FROM t_location WHERE name ~ 'P[uo].*sied(e)?l';
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on t_location  (cost=4.33..19.05 rows=24 width=13)
   Recheck Cond: (name ~ 'P[uo].*sied(e)?l'::text)
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..4.32 rows=24 width=0)
         Index Cond: (name ~ 'P[uo].*sied(e)?l'::text)
(4 rows)
```

11. More fancy LIKE

- An example:

```
test=# SELECT * FROM t_location WHERE name LIKE '%neusi%';
       name
-----------------
 Potzneusiedl
 Markgrafneusiedl
 Gramatneusiedl
(3 rows)

test=# explain SELECT *
                FROM       t_location
                WHERE     name LIKE '%neusi%';
                          QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on t_location  (cost=4.33..19.05 rows=24 width=13)
   Recheck Cond: (name ~~ '%neusi%'::text)
   ->  Bitmap Index Scan on idx_trgm  (cost=0.00..4.32 rows=24 width=0)
         Index Cond: (name ~~ '%neusi%'::text)
(4 rows)
```

12. Indexing MIN / MAX

- An example:

- MIN / MAX works by reading the index from left and right (backward scan)

```
test=# explain SELECT min(relname), max(relname) FROM t_search;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Result  (cost=0.74..0.75 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.27..0.37 rows=1 width=19)
           ->  Index Only Scan using idx_relname on t_search
                       (cost=0.27..29.57 rows=303 width=19)
                 Index Cond: (relname IS NOT NULL)
   InitPlan 2 (returns $1)
     ->  Limit  (cost=0.27..0.37 rows=1 width=19)
           ->  Index Only Scan Backward using idx_relname on
                       t_search t_search_1  (cost=0.27..29.57 rows=303 width=19)
                 Index Cond: (relname IS NOT NULL)
(9 rows)
```

13. BRIN indexes (9.5)


```
test=# CREATE INDEX idx_brin ON t_test USING brin(id);
CREATE INDEX

test=# CREATE INDEX idx_id ON t_test USING btree(id);
CREATE INDEX

test=# SELECT pg_relation_size('idx_id'), pg_relation_size('idx_brin');
 pg_relation_size | pg_relation_size
------------------+------------------
        753713152 |           139264
(1 row)

test=# SELECT count(*) FROM t_test;
  count
----------
 33554432
(1 row)
```

13. BRIN indexes (9.5)

- An example: BRIN indexes are ways smaller but not ideal for OLTP-style queries

```
test=# explain analyze SELECT *
            FROM    t_test
            WHERE id = 4324234 ;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_test  (cost=68.01..72.02 rows=1 width=116)
      (actual time=5.022..6.539 rows=1 loops=1)
   Recheck Cond: (id = 4324234)
   Rows Removed by Index Recheck: 7039
   Heap Blocks: lossy=128
   ->  Bitmap Index Scan on idx_brin  (cost=0.00..68.01 rows=1 width=0)
            (actual time=4.538..4.538 rows=1280 loops=1)
         Index Cond: (id = 4324234)
 Planning time: 0.070 ms
 Execution time: 6.577 ms
(8 rows)
```

Any question?

Thank you for your attention

Any question?