Jsonb «smart» indexing.

Parametrized opclasses

Nikita Glukhov
Postgres Professional

Oleg Bartunov
SAI MSU,
Postgres Professional

PGCONG.RU, Febraury 6, 2018

Index — Silver bullet !

the only weapon that is effective against a werewolf, witch, or other monsters.

Indexes !

- Index is a search tree with tuple pointers in the leaves
- Index has no visibility information (MVCC !)
- Indexes used only for accelerations
- Index scan should produce the same results as sequence scan with filtering
- Indexes can be: **partial** (where price > 0.0), **functional** (to_tsvector(text)), **multicolumn** (timestamp, tsvector)
- Indexes not always useful !
  - Low selectivity
  - Maintainance overhead

Index Encyclopaedia

- Postgres Professional в лице Егора Рогова представляет:
  - Индексы в PostgreSQL — 1,2
    https://habrahabr.ru/company/postgrespro/blog/326096/
    https://habrahabr.ru/company/postgrespro/blog/326106/
  - Hash индекс
    https://habrahabr.ru/company/postgrespro/blog/328280/
  - B-tree
    https://habrahabr.ru/company/postgrespro/blog/330544/
  - GiST
    https://habrahabr.ru/company/postgrespro/blog/333878/

Index Encyclopaedia

- Postgres Professional в лице Егора Рогова представляет:
  - SPGiST
    https://habrahabr.ru/company/postgrespro/blog/337502/
  - GIN
    https://habrahabr.ru/company/postgrespro/blog/340978/
  - RUM
    https://habrahabr.ru/company/postgrespro/blog/343488/
  - BRIN
    https://habrahabr.ru/company/postgrespro/blog/346460/

Extending Indexing infrastructure

- Opclasses have «hardcoded» constants
  - Let user to define these constants for specific data
- Indexing of non-atomic data (arrays, json[b], tsvector,…)
  - Specify what to index — partial index only filters rows
- Use different algorithms to index
  - Specify what to use depending on data

Opclass parameters: syntax

- Parenthized parameters added after column's opclass. Default opclass can be specified with DEFAULT keyword:

```
CREATE INDEX idx ON tab USING am (
    {expr {DEFAULT | opclass} ({name=value} [,…])} [,…]
) …
```

```
CREATE INDEX ON small_arrays USING gist (
     arr gist__intbig_ops(siglen=32),
     arr DEFAULT (num_ranges = 100)
);
```

Opclass parameters: implementation

- Parameters are stored in `text[]` column `indoptions` of `pg_catalog.pg_index`
- i-th element of `indoptions[]` is a `text` array of parameters of i-th index column serialized into a string.
- Each parameter is stored as 'name=value' text string.

Opclass parameters: implementation

- Parameters are stored in `text[]` column `indoptions` of `pg_catalog.pg_index`

```
SELECT (
      SELECT array_agg(opcname) opclasses
      FROM unnest(indclass::int[]) opcid, pg_opclass opc
      WHERE opc.oid=opcid), indoptions
FROM pg_index
WHERE indoptions IS NOT NULL;
            opclasses                  |               indoptions
------------------------------------+------------------------------------
 {jsonb_path_ops}                      | {"{paths=$.tags[*].term}"}
 {gist__intbig_ops,gist__int_ops} | {"{siglen=32}","{num_ranges=100}"}
(3 rows)
```

Opclass parameters: implementation

- Each access method supporting opclass parameters specifies `amopclassoptions` routine for transformation of `text[]` parameters datum into a binary bytea structure which will be cached in `RelationData` and `IndexOptInfo` structures:

```
typedef bytea *(*amopclassoptions_function) (
    Relation index, AttrNumber colnum,
    Datum indoptions, bool validate
);
```

Opclass parameters: implementation

- Access method which wants simply to delegate parameters processing to one of column opclass's support functions can use `index_opclass_options_generic()` subroutine in its `amopclassoptions` implementation.

```
bytea *index_opclass_options_generic(
   Relation relation, AttrNumber attnum,
   uint16 procnum, Datum indoptions, bool validate);
```

- This support functions must have the following signature:

```
internal (options internal, validate bool)
```

Opclass options passed as a `text[]` datum.

Returned pointer to bytea.

Opclass parameters: implementation

- Example: contrib/intarray

```
CREATE FUNCTION g_intbig_options(internal, boolean)
RETURNS internal
AS 'MODULE_PATHNAME', 'g_intbig_options'
LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;

ALTER OPERATOR FAMILY gist__intbig_ops USING gist
ADD FUNCTION 10 (_int4) g_intbig_options (internal, boolean);
```

PostgreSQL extendability

It is imperative that a user be able to construct new access methods to provide efficient access to instances of nontraditional base types

Michael Stonebraker, Jeff Anton, Michael Hirohama.

Extendability in POSTGRES , IEEE Data Eng. Bull. 10 (2) pp.16-23, 1987

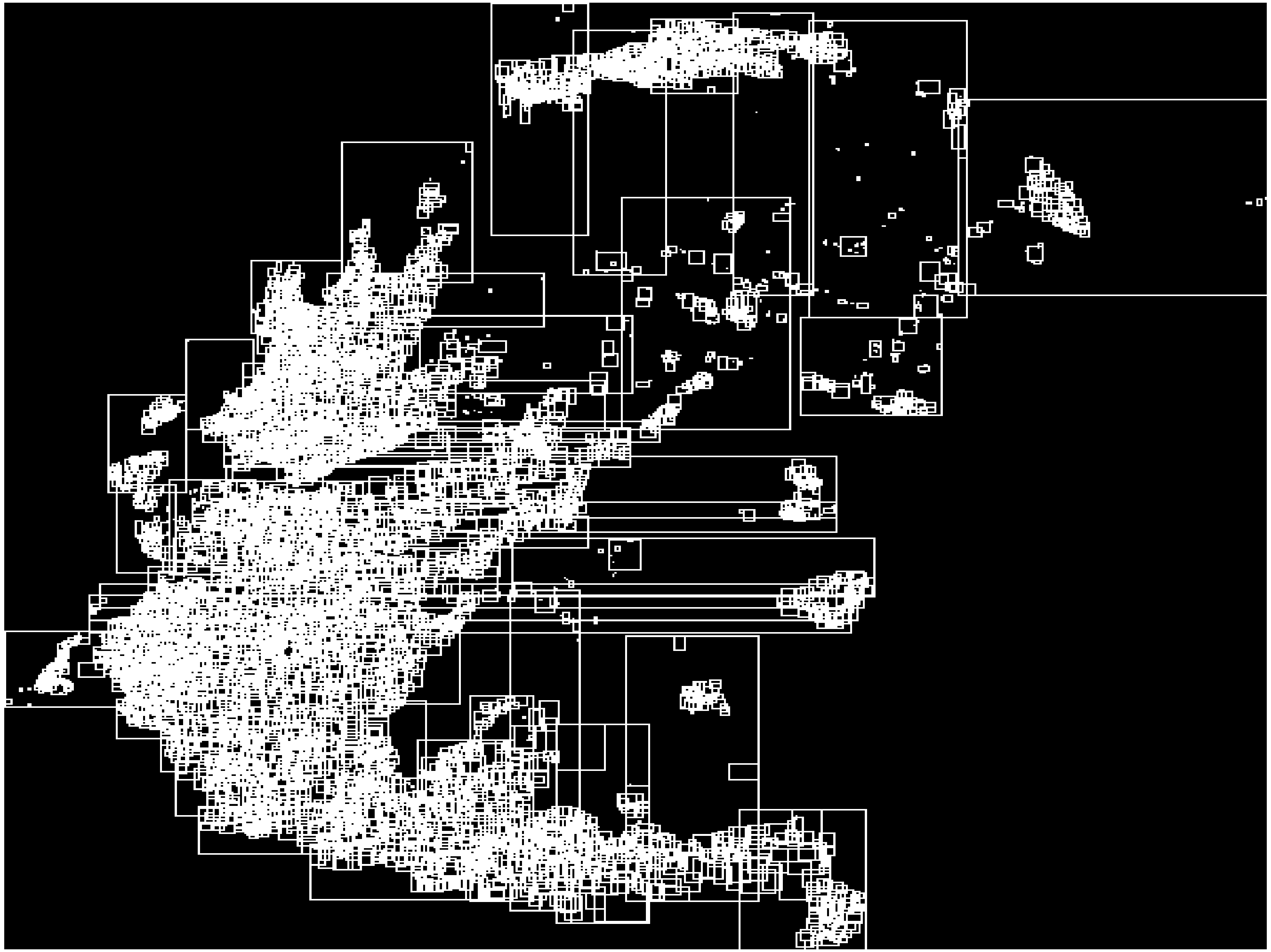
Расширяемость PostgreSQL: GiST

- Generalized Search Tree (GiST)
  J. M. Hellerstein, J. F. Naughton, and Avi Pfeffer. "Generalized search trees for database systems.", VLDB 21, 1995
  - AM рассматривается как иерархия предикатов, в которой каждый предикат выполняется для всех подузлов этой иерархии
  - Шаблон (template) для реализации новых AM
- GiST предоставляет методы
  - навигации по дереву, эффективный knn
  - Обновления дерева

Расширяемость PostgreSQL: GiST

- Конкурентность и восстановление после сбоев
- Поддерживает расширяемый набор запросов (в отличие от фиксированных операций сравнения B-tree)
- GiST позволяет реализовать новый AM эксперту в области данных
- Новые типы данных обладают производительностью (индексный доступ, конкурентность) и надежностью (протокол логирования), как и встроенные типы

GiST — Rtree example (Greece)



Understanding GiST (array example)

- Intarray -Access Method for array of integers
  - Operators overlap, contains

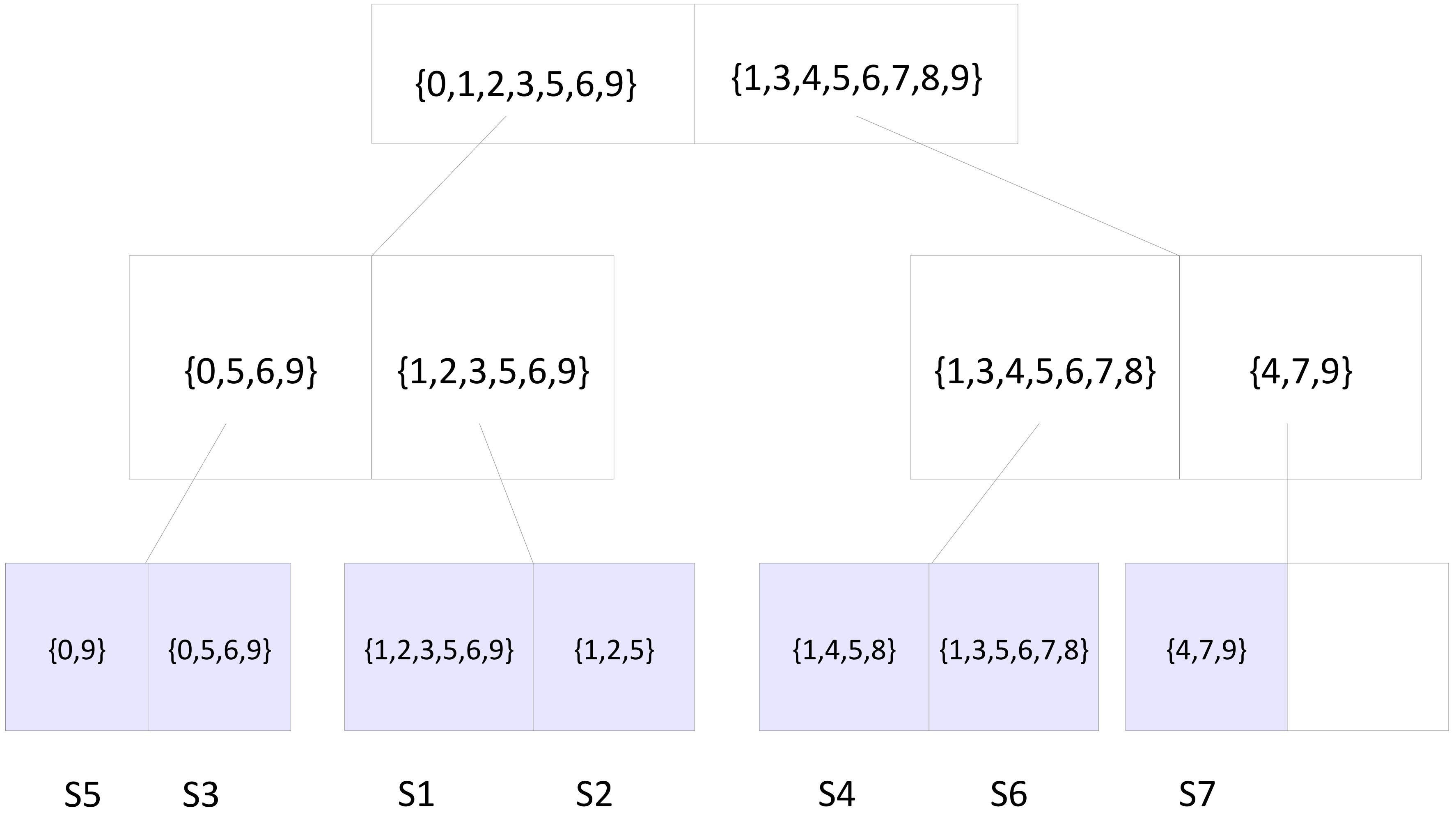
S1 = {1,**2**,3,5,6,**9**}
S2 = {1,**2**,5}
S3 = {0,5,6,**9**}
S4 = {1,4,5,8}
S5 = {0,9}
S6 = {3,5,6,7,8}
S7 = {4,7,**9**}

Q = {**2,9**}

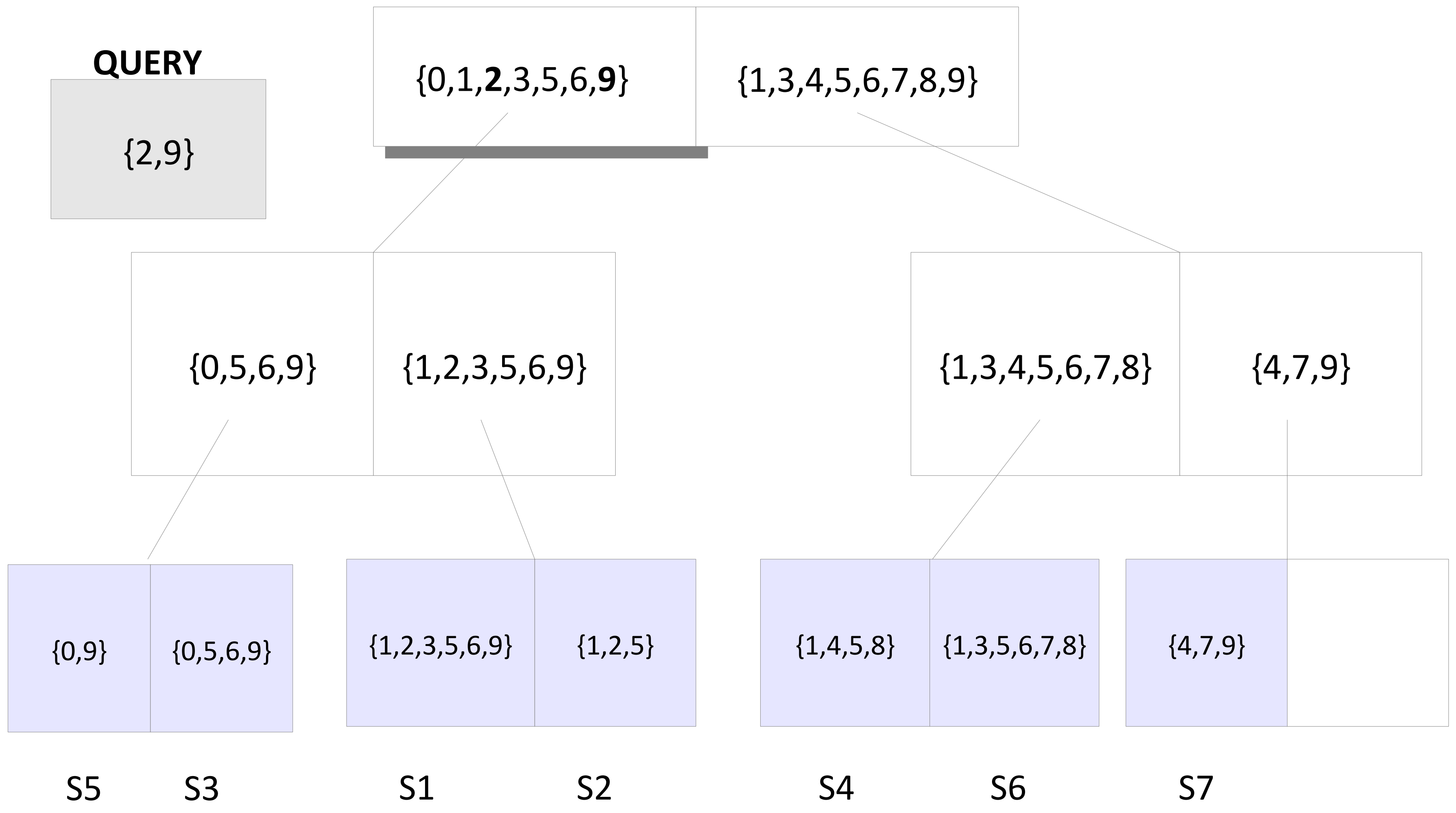
"THE RD-TREE: AN INDEX STRUCTURE FOR SETS", Joseph M. Hellerstein

RD-Tree

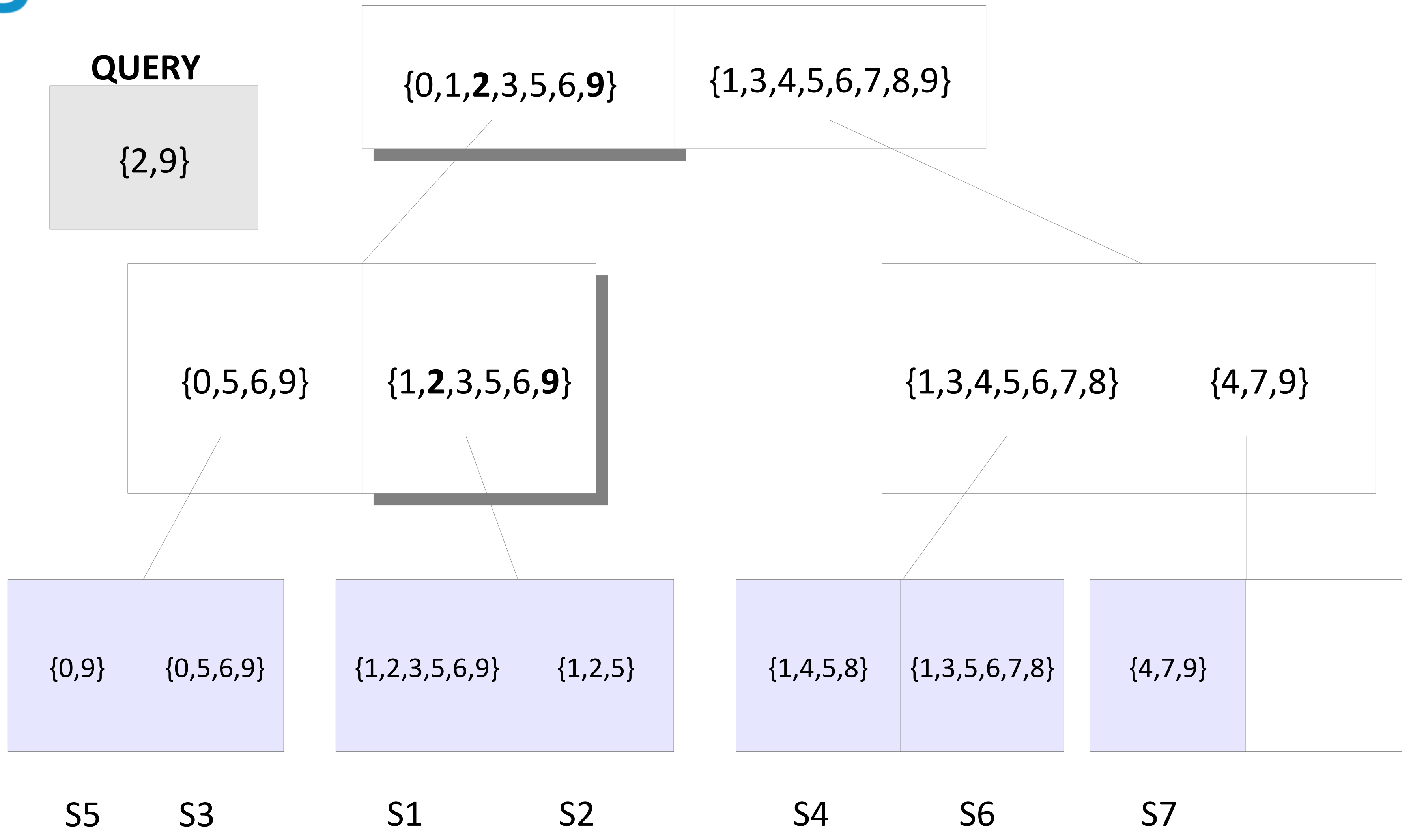
Containment Hierarchy



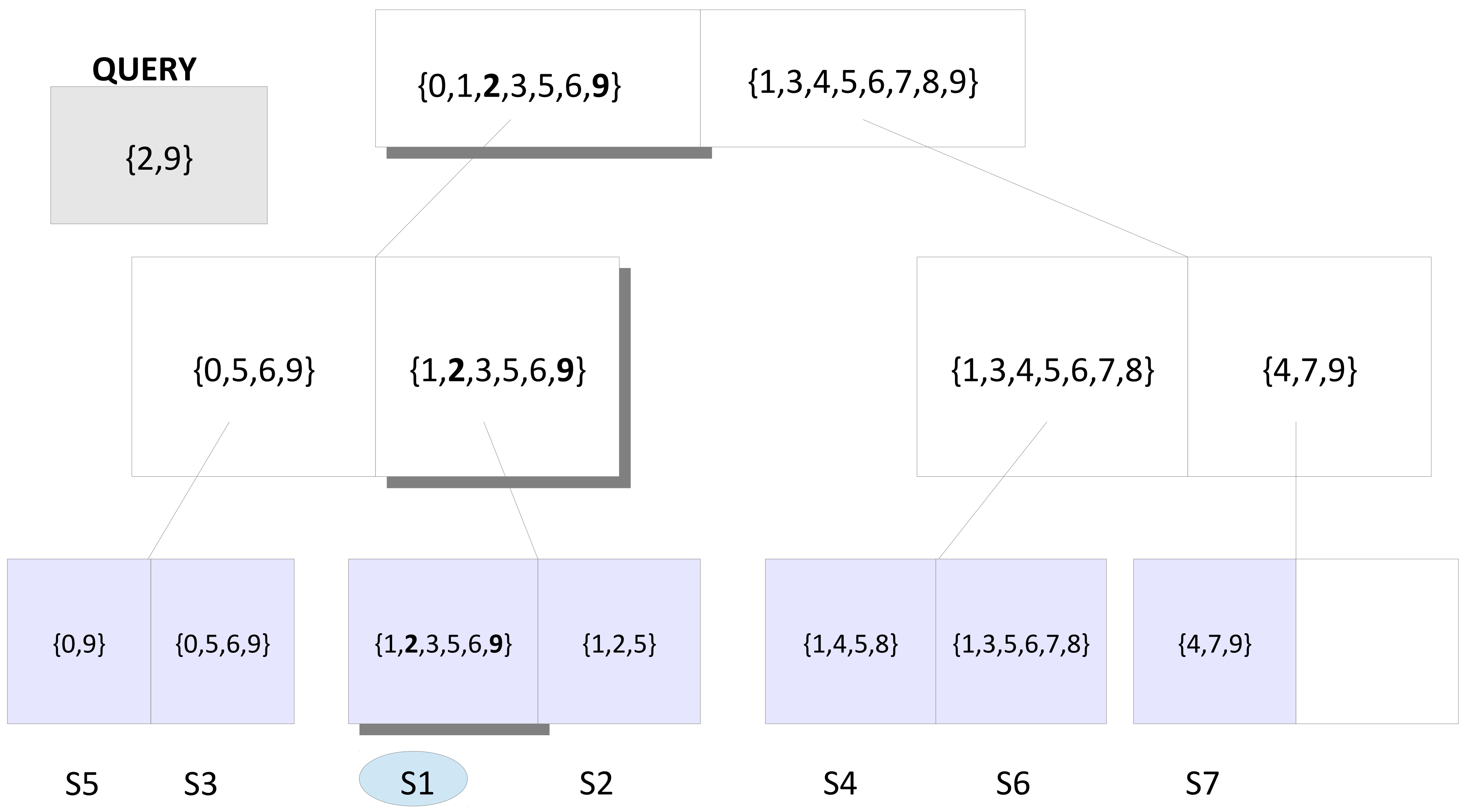
RD-Tree


QUERY
{2,9}
{0,1,**2**,3,5,6,**9**}
{1,3,4,5,6,7,8,9}
{0,5,6,9}
{1,2,3,5,6,9}
{1,3,4,5,6,7,8}
{4,7,9}
{0,9}
{0,5,6,9}
{1,2,3,5,6,9}
{1,2,5}
{1,4,5,8}
{1,3,5,6,7,8}
{4,7,9}
S5
S3
S1
S2
S4
S6
S7

RD-Tree


QUERY
{2,9}
{0,1,**2**,3,5,6,**9**}
{1,3,4,5,6,7,8,9}
{0,5,6,9}
{1,**2**,3,5,6,**9**}
{1,3,4,5,6,7,8}
{4,7,9}
{0,9}
{0,5,6,9}
{1,2,3,5,6,9}
{1,2,5}
{1,4,5,8}
{1,3,5,6,7,8}
{4,7,9}
S5
S3
S1
S2
S4
S6
S7

RD-Tree



QUERY
{2,9}
{0,1,**2**,3,5,6,**9**}
{1,3,4,5,6,7,8,9}
{0,5,6,9}
{1,**2**,3,5,6,**9**}
{1,3,4,5,6,7,8}
{4,7,9}
{0,9}
{0,5,6,9}
{1,**2**,3,5,6,**9**}
{1,2,5}
{1,4,5,8}
{1,3,5,6,7,8}
{4,7,9}
S5
S3
S1
S2
S4
S6
S7

FTS Index (GiST): RD-Tree

- Word signature — words hashed to the specific position of '1'

  w1 -> S1: 01000000 Document: w1 w2 w3

  w2 -> S2: 00010000

  w3 -> S3: 10000000

- Document (query) signature — superposition (bit-wise OR) of signatures

  S: 11010000

- Bloom filter

  Q1: 00000001 – exact not

  Q2: 01010000 - may be contained in the document, **false drop**

- Signature is a lossy representation of document
  - + fixed length, compact, + fast bit operations
  - - lossy (false drops), - saturation with #words grows

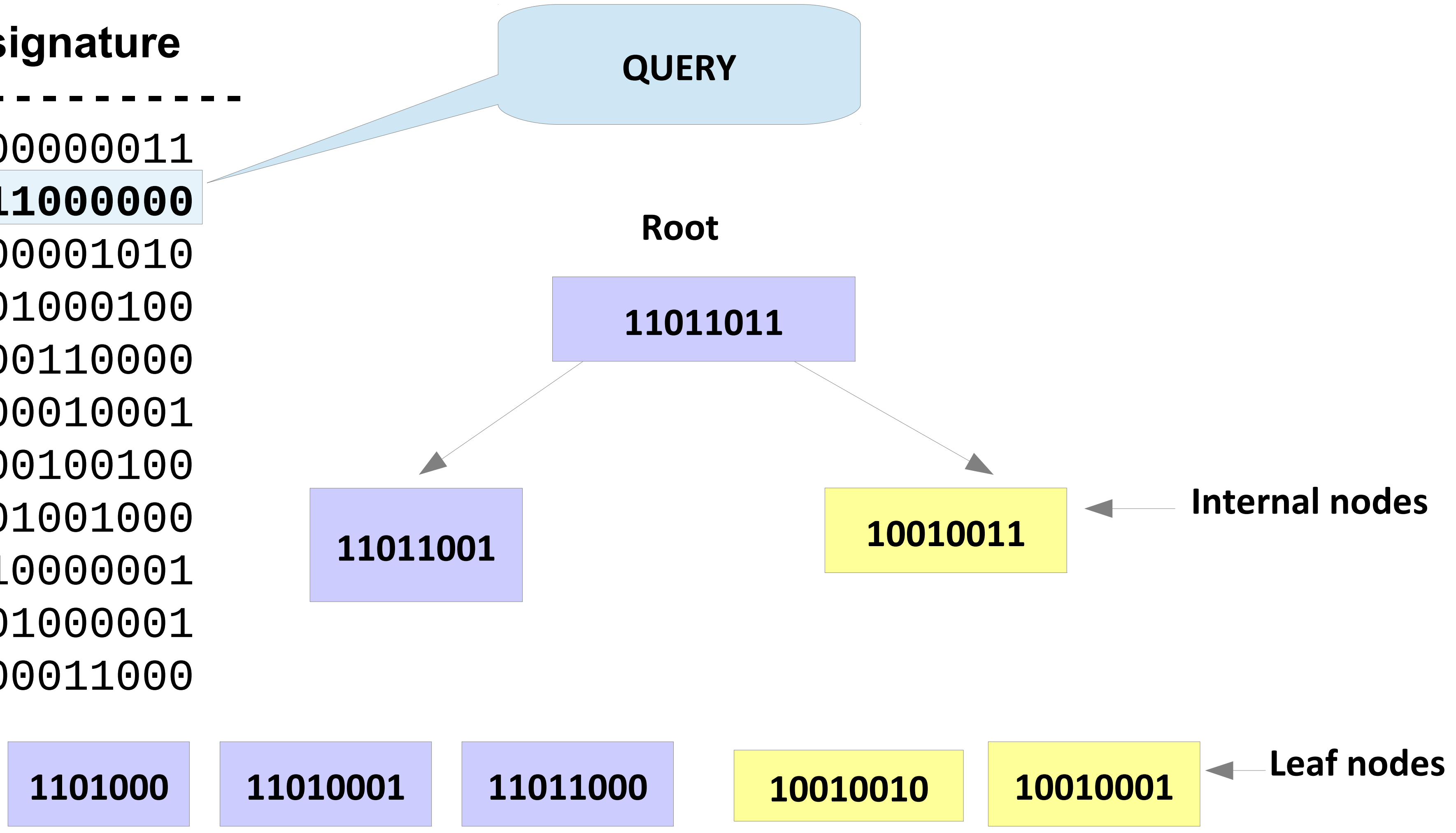
FTS Index (GiST): RD-Tree

- Latin proverbs

```
id |          proverb
---+-----------------------
 1 | Ars longa, vita brevis
 2 | Ars vitae
 3 | Jus vitae ac necis
 4 | Jus generis humani
 5 | Vita nostra brevis
```

FTS Index (GiST): RD-Tree

```
  word    | signature
----------+-----------
 ac       | 00000011
 ars      | 11000000
 brevis   | 00001010
 generis  | 01000100
 humani   | 00110000
 jus      | 00010001
 longa    | 00100100
 necis    | 01001000
 nostra   | 10000001
 vita     | 01000001
 vitae    | 00011000
```



RD-Tree (GiST)

```
 id |          proverb           | signature
----+----------------------------+------------
  1 | Ars longa, vita brevis     | 11101111
  2 | Ars vitae                  | 11011000
  3 | Jus vitae ac necis         | 01011011
  4 | Jus generis humani         | 01110101
  5 | Vita nostra brevis         | 11001011
```

False drop

Opclass parameters: GiST

- Parametrized GiST opclass should specify optional 10th (GIST_OPCLASSOPT_PROC) support function with signature:

  `internal (options internal, validate bool)`

- Returned parsed bytea * with parameters will be passed in all support functions in the last argument.

Opclass parameters: GiST examples

- core:
  tsvector_ops(siglen)
- contrib/intarray:
  gist__int_ops(num_ranges)

  gist__intbig_ops(siglen)
- contrib/ltree:

  gist_ltree_ops(siglen)

  gist__ltree_ops(siglen)
- contrib/pg_trgm:

  gist_trgm_ops(siglen)

Opclass parameters: intarray

- contrib/intarray — NULL-free arrays of integers (https://www.postgresql.org/docs/current/static/intarray.html)

  Пример:

  Int[] && int[] - overlap
  Поиск товаров из определенных категорий
- Индексы

  gist__int_ops(num_ranges)

  gist__intbig_ops(siglen) - используйте этот опкласс.

Opclass parameters: intarray

```
SELECT i AS id,
(select  array_agg(round(random() * (100 - 1)) + 1)::int[]
FROM generate_series (0,  (random()*100 + i * 0)::int))
AS arr INTO arrays FROM generate_series(1,1000000) i;

                Table "public.arrays"  – 255MB size
 Column |    Type     | Collation | Nullable | Default
--------+-------------+-----------+----------+---------
 id     | integer     |           |          |
 arr    | integer[]   |           |          |
Indexes:
    "arrays_arr_idx" gist (arr gist__intbig_ops)  -- siglen = 252
    "arrays_arr_idx1" gist (arr gist__intbig_ops (siglen='64'))
    "arrays_arr_idx2" gist (arr gist__intbig_ops (siglen='32'))
```

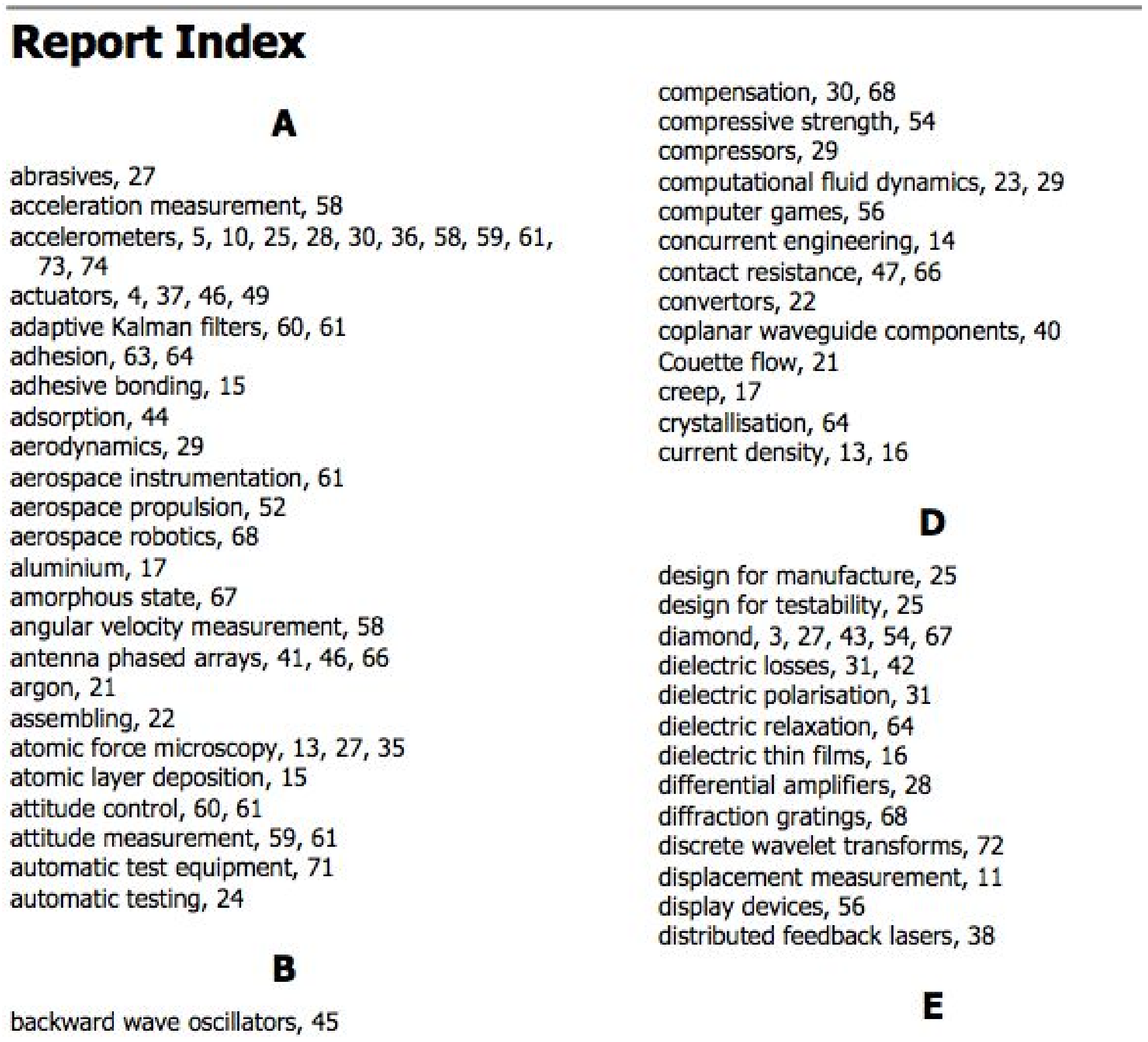
Opclass parameters: intarray

```
                               List of relations
 Schema |        Name        | create |  Size    | Q1   | Q2
--------+------------------+---------+---------+---------+-
 public | arrays_arr_idx   | 28  sec| 451 MB | 641 | 387
 public | arrays_arr_idx1 | 17  sec| 143 MB | 535 | 164
 public | arrays_arr_idx2 | 15  sec| 86 MB  | 527 |  73
(3 rows)

Q1: SELECT count(*) FROM arrays WHERE arr @> '{1,2,3}';
Q1: SELECT count(*) FROM arrays WHERE arr <@ '{1,2,3}';
```

GIN
Generalized Inverted Index

GIN



Report Index

A

abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B

backward wave oscillators, 45

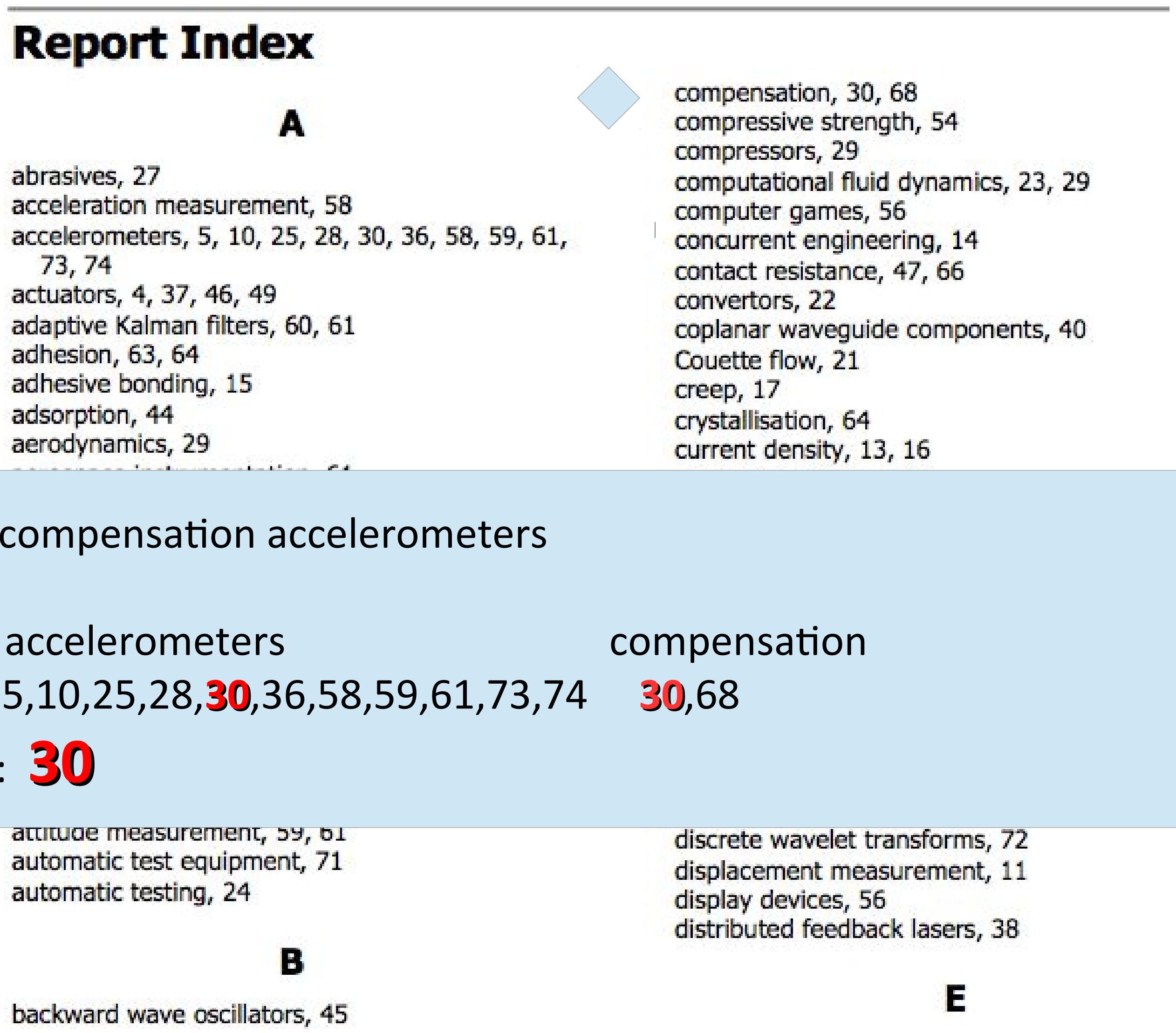
compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
convertors, 22
coplanar waveguide components, 40
Couette flow, 21
creep, 17
crystallisation, 64
current density, 13, 16

D

design for manufacture, 25
design for testability, 25
diamond, 3, 27, 43, 54, 67
dielectric losses, 31, 42
dielectric polarisation, 31
dielectric relaxation, 64
dielectric thin films, 16
differential amplifiers, 28
diffraction gratings, 68
discrete wavelet transforms, 72
displacement measurement, 11
display devices, 56
distributed feedback lasers, 38

E

GIN



Report Index

A

abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61,
  73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29

attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B

backward wave oscillators, 45

compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
convertors, 22
coplanar waveguide components, 40
Couette flow, 21
creep, 17
crystallisation, 64
current density, 13, 16

discrete wavelet transforms, 72
displacement measurement, 11
display devices, 56
distributed feedback lasers, 38

E

QUERY: compensation accelerometers

INDEX: accelerometers — 5,10,25,28,**30**,36,58,59,61,73,74

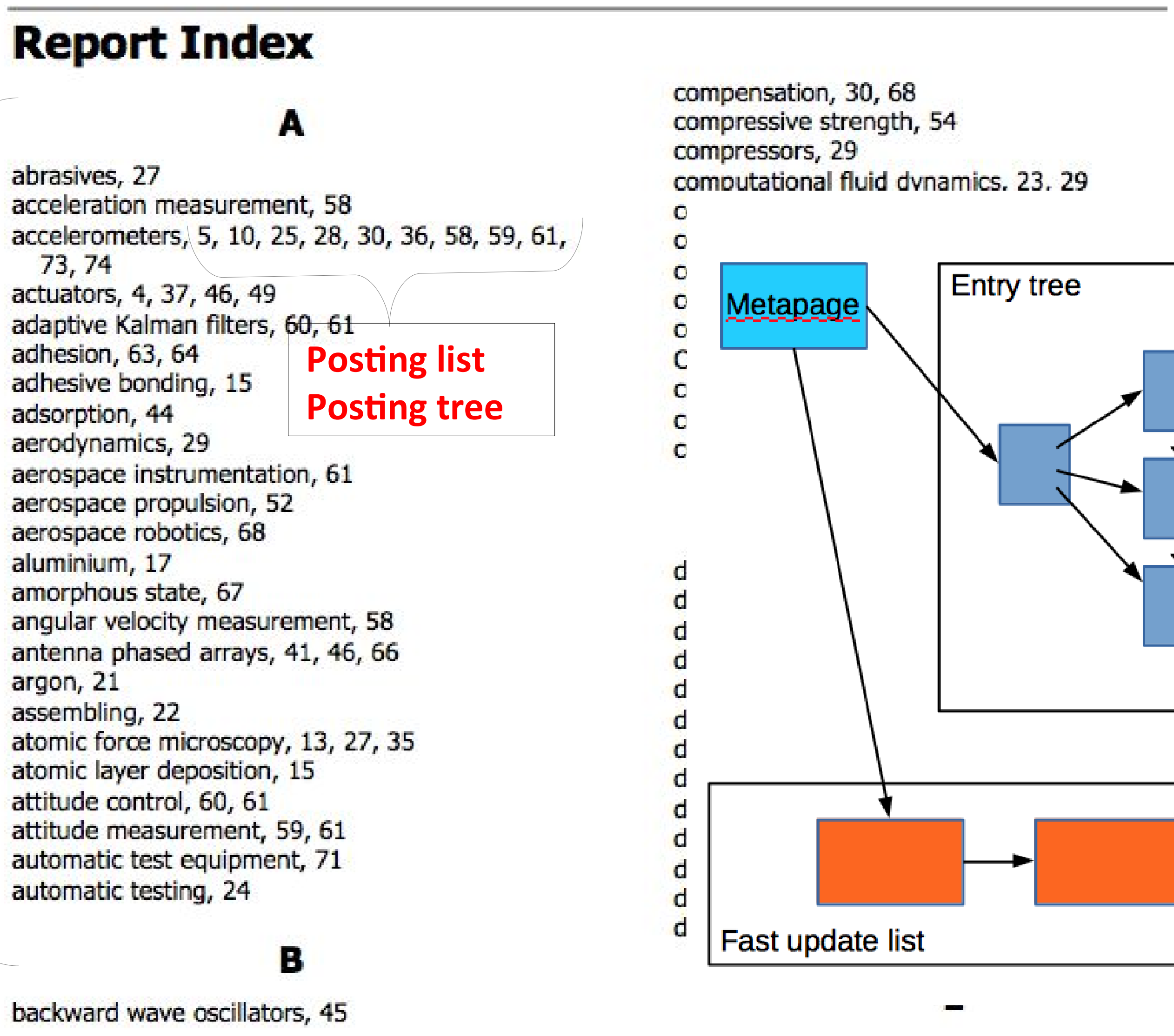
compensation — **30**,68

RESULT: **30**

No positions in index ! (use RUM for storing positions)

Inverted Index in PostgreSQL

E N T R Y T R E E



Report Index

A

abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

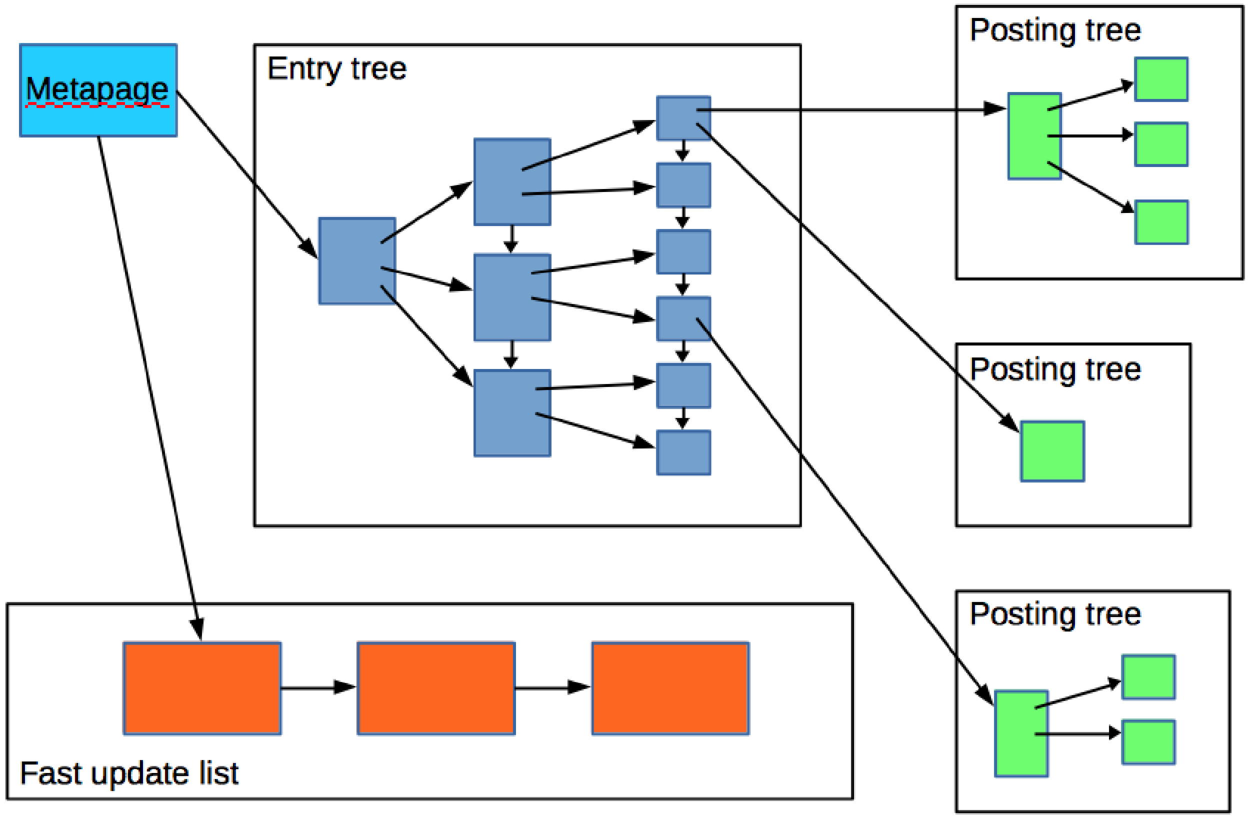
B

backward wave oscillators, 45

compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
c
c
c
c
c
C
c
c
c

d
d
d
d
d
d
d
d
d
d
d
d
d

Posting list
Posting tree



GIN

- Internal structure is basically just a B-tree
  - Optimized for storing a lot of duplicate keys
  - Duplicates are ordered by heap TID
- Interface supports indexing more than one key per indexed value
  - Full text search: “foo bar” → “foo”, “bar”
- Bitmap scans only

GIN Index

Demo collections – latin proverbs

```
 id |         proverb
----+-----------------------------
  1 | Ars longa, vita brevis
  2 | Ars vitae
  3 | Jus vitae ac necis
  4 | Jus generis humani
  5 | Vita nostra brevis
```

GIN Index

Inverted Index

Entry tree

Posting tree

```
  word    | posting
----------+----------
 ac       | {3}
 ars      | {1,2}
 brevis   | {1,5}
 generis  | {4}
 humani   | {4}
 jus      | {3,4}
 longa    | {1}
 necis    | {3}
 nostra   | {5}
 vita     | {1,5}
 vitae    | {2,3}
```

Fast search

Slow update

Opclass parameters: GIN

- Parametrized GIN opclass should specify optional 7th (`GIN_OPCLASSOPTIONS_PROC`) support function with signature:

  ```
  internal (options internal, validate bool)
  ```

- Returned parsed bytea * with parameters will be passed in all support functions in the last argument.

Opclass parameters: GIN examples

- core:
  tsvector_ops(weights)

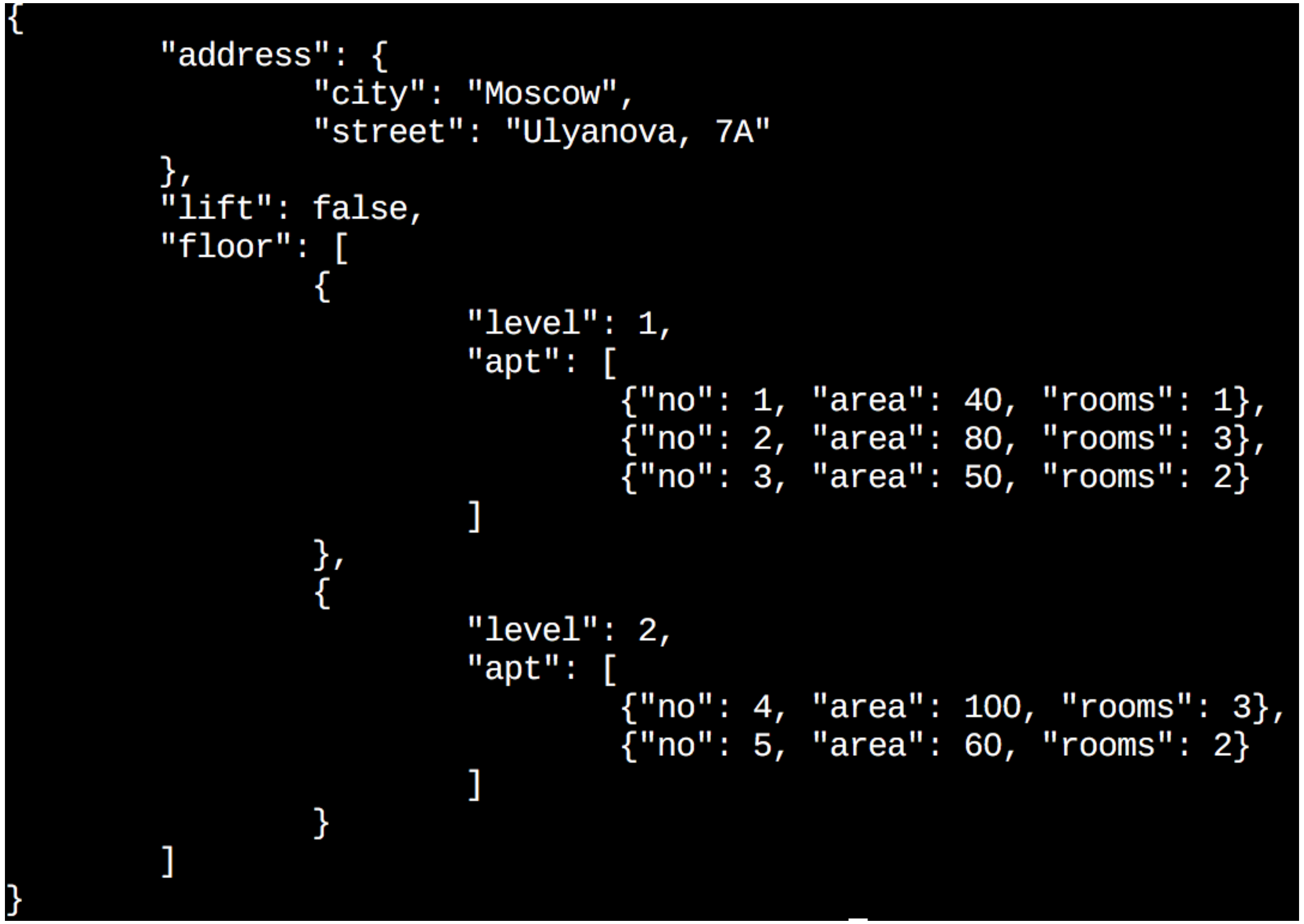
- core (used jsonpath type from SQL/JSON branch):
  jsonb_ops(projection)
  jsonb_path_ops(projection)

SQL/JSON in PostgreSQL

- SQL-2016 path language specifies the parts (the projection) of JSON data to be retrieved by path engine for SQL/JSON functions.
- **Jsonpath —** the binary data type for SQL/JSON path expression to effective query JSON data.

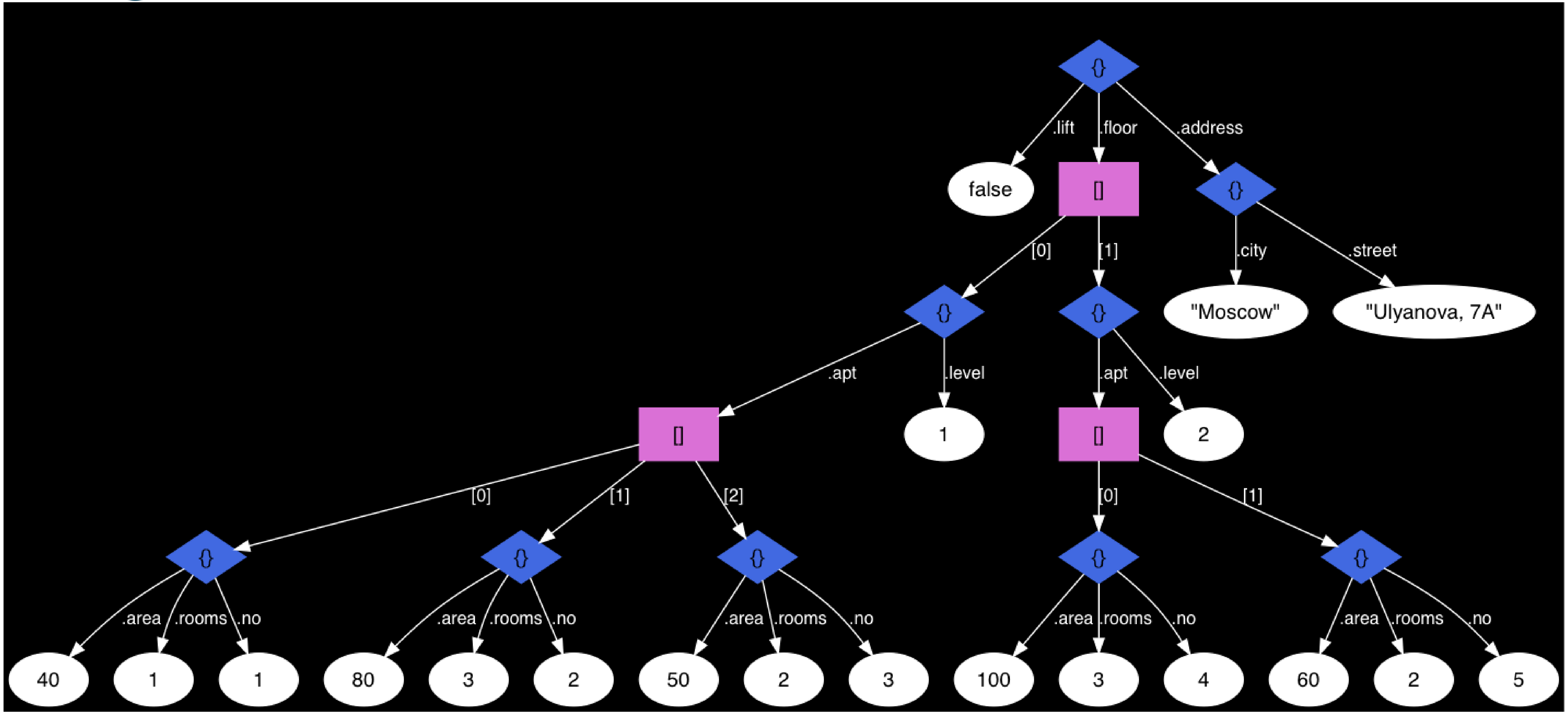
```
SELECT JSON_QUERY(js,
   '$.floor[*] ? (@.level >1).apt[*] ? (@.area>$min && @.area < $max).no'
                    PASSING 40 AS min, 90 AS max )
FROM house;
```

Visual guide on jsonpath

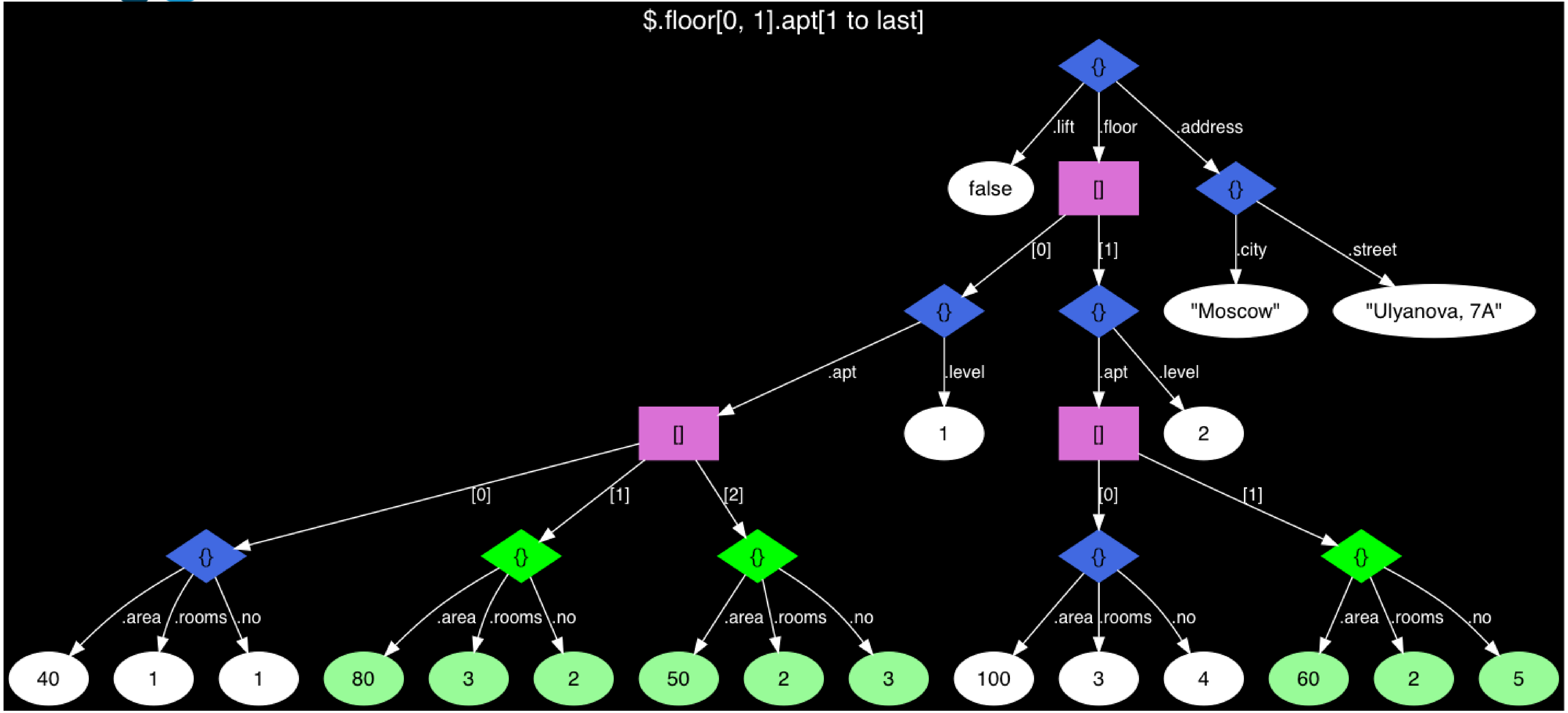


```
{
        "address": {
                "city": "Moscow",
                "street": "Ulyanova, 7A"
        },
        "lift": false,
        "floor": [
                {
                        "level": 1,
                        "apt": [
                                {"no": 1, "area": 40, "rooms": 1},
                                {"no": 2, "area": 80, "rooms": 3},
                                {"no": 3, "area": 50, "rooms": 2}
                        ]
                },
                {
                        "level": 2,
                        "apt": [
                                {"no": 4, "area": 100, "rooms": 3},
                                {"no": 5, "area": 60, "rooms": 2}
                        ]
                }
        ]
}
```

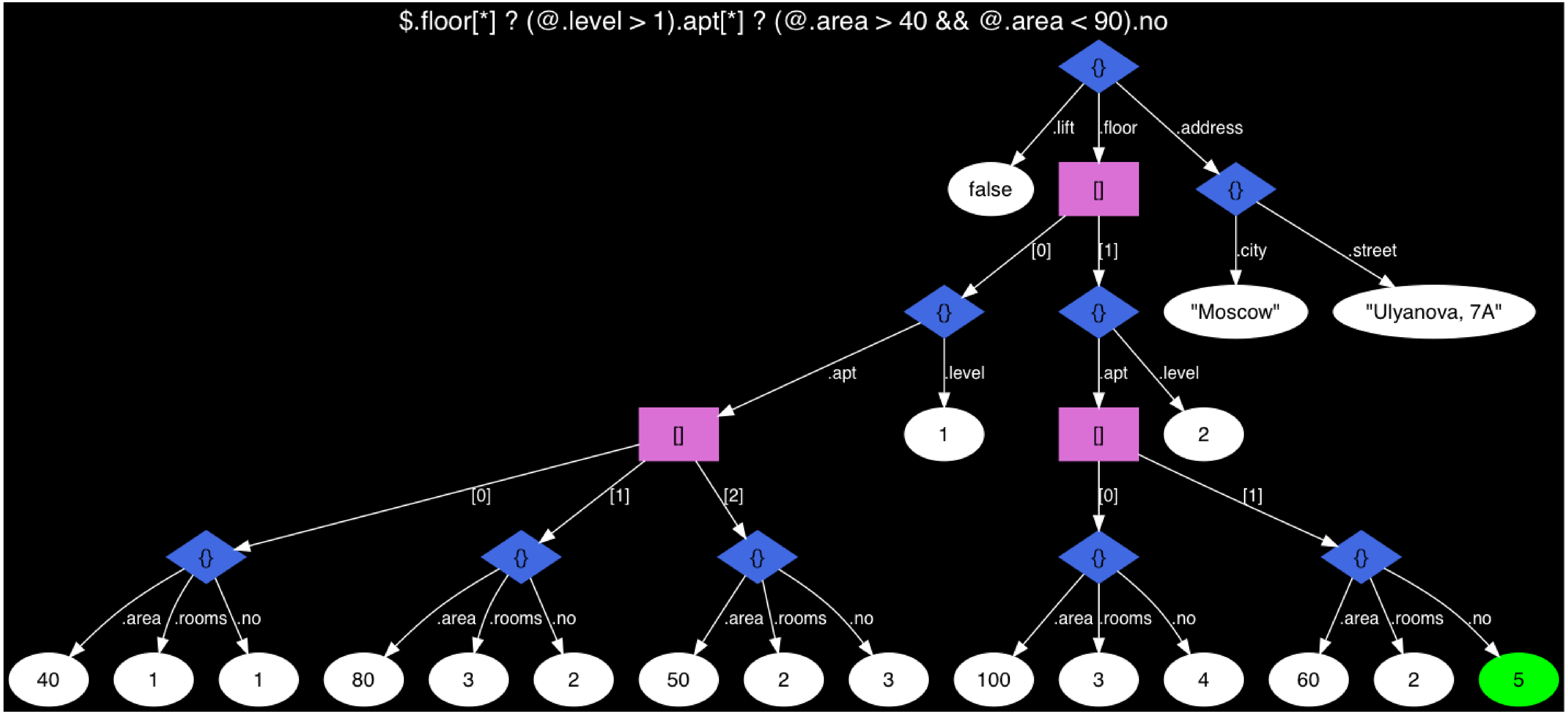
2-floors house



$.floor[0, 1].apt[1 to last]



$.floor[*]?(@.level >1).apt[*]?(@.area>40 && @.area < 90).no



$.floor[0, 1].apt[1 to last]

```
SELECT JSON_QUERY(js, '$.floor[0, 1].apt[1 to last]' WITH WRAPPER) FROM house;
                                                   ?column?
--------------------------------------------------------------------------------------
 [{"no": 2, "area": 80, "rooms": 3}, {"no": 3, "area": 50, "rooms": 2},
  {"no": 5, "area": 60, "rooms": 2}]
(1 row)
```

JSON_TABLE — relational view of json

```
SELECT  apt.*
FROM
  house,
  JSON_TABLE(js, '$.floor[0, 1]' COLUMNS (
    level int,
    NESTED PATH '$.apt[1 to last]' COLUMNS (
      no int,
      area int,
      rooms int
    )
  )) apt;
 level | no | area | rooms
-------+----+------+-------
     1 |  2 |   80 |     3
     1 |  3 |   50 |     2
     2 |  5 |   60 |     2
(3 rows)
```

Opclass parameters: jsonb

- Added `projection` parameter to GIN `jsonb_ops` and `jsonb_path_ops` for specification of indexed fragments of JSON document.
- `projection` parameter is a string in standard SQL/JSON path format. We use our jsonpath syntax extension for joining path expressions into sequences with comma symbol:
  `jsonpath '$.a[*], $.b, $.c[*].d'`
- Each indexed path must start with $ and only the following path elements are supported now:
  `.key  .*  .**{1,3}`
  `[*]`
  `[3, 5 to 7]` (constant indices only)

Opclass parameters: jsonb

- Indexed path specifications can be strict or lax (default).
  The following projections are equivalent due to automatic array wrapping/unwrapping in lax mode:

```
projection='lax $.a[*].b'
projection='strict $.a.b, $[*].a.b, $.a[*].b, $[*].a[*].b,
          $.a.b[*], $[*].a.b[*], $.a[*].b[*], $[*].a[*].b[*]'
```

- Sequential scan is used if the query does not emit any indexed entries or contains not-indexed path in OR-ed predicate:

```
projection='$.a'
```

`js @~ '$.b == 1'` => sequential scan
`js @~ '$.a == 1 || $.b == 2'` => sequential scan
`js @~ '$.a == 1 && $.b == 2'` => index scan of `$.a` with recheck which filters $.b

Opclass parameters: jsonb bookmarks examples

- JSON document structure: {

```
 "id": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422#mcasas1",
 "author": "mcasas1",
 "title": "TheaterMania",
 "updated": "Tue, 08 Sep 2009 23:28:55 +0000",
 "tags": [
   {"term": "NYC", "scheme": "http://delicious.com/mcasas1/", "label": null}
 ],
 "link": "http://www.theatermania.com/broadway/",
 "links": [{"href": "http://www.theatermania.com/broadway/",
            "type": "text/html", "rel": "alternate"}],
 "comments": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422",
 "guidislink": false,
 "title_detail": {
    "base": "http://feeds.delicious.com/v2/rss/recent?min=1&count=100",
    "type": "text/plain", "language": null, "value": "TheaterMania"
 },
 "source": {},
 "wfw_commentrss": "http://feeds.delicious.com/v2/rss/url/b5b3cbf9a9176fe43c27d7b4af94a422"
}
```

Opclass parameters: jsonb bookmarks examples

- We want to find bookmarks by tag terms
- Full and selective jsonb indices:

```
CREATE INDEX bookmarks_full_idx ON bookmarks USING
  gin(js jsonb_ops);

CREATE INDEX bookmarks_full_path_idx ON bookmarks USING
  gin(js jsonb_path_ops);

CREATE INDEX bookmarks_selective_idx ON bookmarks USING
  gin(js jsonb_ops(projection='strict $.tags[*].term'));

CREATE INDEX bookmarks_selective_path_idx ON bookmarks USING
  gin(js jsonb_path_ops(projection='strict $.tags[*].term'));
```

Opclass parameters: jsonb bookmarks examples

- Functional indices on SQL/JSON functions and operators:

```
CREATE INDEX bookmarks_oper_idx ON bookmarks USING
  gin((js @# '$.tags.term'));

CREATE INDEX bookmarks_oper_path_idx ON bookmarks USING
  gin((js @# '$.tags.term') jsonb_path_ops);

CREATE INDEX bookmarks_json_query_jsonb_idx ON bookmarks USING
  gin(JSON_QUERY(js, '$.tags.term' WITH WRAPPER));

CREATE INDEX bookmarks_json_query_jsonb_path_idx ON bookmarks USING
  gin(JSON_QUERY(js, '$.tags.term' WITH WRAPPER) jsonb_path_ops);

CREATE INDEX bookmarks_json_query_texta_idx ON bookmarks USING
  gin(JSON_QUERY(js, '$.tags.term' RETURNING text[] WITH WRAPPER)
COLLATE "C");
```

Opclass parameters: jsonb bookmarks examples

- Functional indices on plain jsonb functions:

```
CREATE FUNCTION bookmark_tags(bookmark jsonb) RETURNS jsonb
  AS $$ SELECT jsonb_agg(tag->'term')
        FROM jsonb_array_elements($1->'tags') tag(tag)
  $$ LANGUAGE sql IMMUTABLE;

CREATE FUNCTION bookmark_tags_text(bookmark jsonb) RETURNS text[]
  AS $$ SELECT array_agg(tag->>'term')
        FROM jsonb_array_elements($1->'tags') tag(tag)
  $$ LANGUAGE sql IMMUTABLE;


CREATE INDEX bookmarks_func_jsonb_idx ON bookmarks USING
  gin(bookmark_tags(js));
CREATE INDEX bookmarks_func_jsonb_path_idx ON bookmarks USING
  gin(bookmark_tags(js) jsonb_path_ops);
CREATE INDEX bookmarks_func_texta_idx ON bookmarks USING
  gin(bookmark_tags_text(js) COLLATE "C");
```

Opclass parameters: jsonb bookmarks examples

```
              Index                 |  Size   | Build time
------------------------------------+---------+------------
 bookmarks_full_idx                 | 629 MB  |  183.1 s
 bookmarks_full_path_idx            | 300 MB  |   98.5 s
 bookmarks_selective_idx            |  32 MB  |    8.0 s
 bookmarks_selective_path_idx       |  33 MB  |    4.5 s
 bookmarks_oper_idx                 |  35 MB  |    6.5 s
 bookmarks_oper_path_idx            |  33 MB  |    4.5 s
 bookmarks_json_query_jsonb_idx     |  35 MB  |    7.3 s
 bookmarks_json_query_jsonb_path_idx|  33 MB  |    5.4 s
 bookmarks_json_query_texta_idx     |  34 MB  |    7.5 s
 bookmarks_func_jsonb_idx           |  35 MB  |   18.8 s
 bookmarks_func_jsonb_path_idx      |  33 MB  |   17.2 s
 bookmarks_func_texta_idx           |  34 MB  |   15.1 s
```

Как отлаживать индексы ?

- Много индексов, тяжело и долго переделывать, можно один раз сделать и показывать планеру только нужный.
- Plantuner - http://www.sai.msu.su/~megera/wiki/plantuner

  Пример использования https://obartunov.livejournal.com/197604.html

Opclass parameters: jsonb bookmarks examples

Queries for full or selective index
(performance is the same for them):

-
  ```
  SELECT COUNT(*) FROM bookmarks
  WHERE js @> '{"tags": [{"term": "NYC"}]}';
  ```
  ```
  0.664 ms jsonb_path_ops
  7.399 ms jsonb_ops
  ```

-
  ```
  SELECT COUNT(*) FROM bookmarks
  WHERE js @~ '$.tags.term == "NYC"';
  ```
  ```
  0.722 ms jsonb_path_ops
  7.500 ms jsonb_ops
  ```

-
  ```
  SELECT COUNT(*) FROM bookmarks
  WHERE js @~ 'strict $.tags[*].term == "NYC"';
  ```
  ```
  0.719 ms jsonb_path_ops
  7.414 ms jsonb_ops
  ```

-
  ```
  SELECT COUNT(*) FROM bookmarks
  WHERE js @? '$.tags ? (@.term == "NYC")';
  ```
  ```
   0.685 ms jsonb_path_ops
  14.978 ms jsonb_ops
  ```

Opclass parameters: jsonb bookmarks examples

Queries for functional indices:

- `SELECT COUNT(*) FROM bookmarks WHERE` **`js @# '$.tags.term' @> '"NYC"'`**`;`
- `SELECT COUNT(*) FROM bookmarks WHERE` **`JSON_QUERY(js, '$.tags.term' WITH WRAPPER) @> '"NYC"'`**`;`
- `SELECT COUNT(*) FROM bookmarks WHERE` **`JSON_QUERY(js, '$.tags.term' RETURNING text[] WITH WRAPPER) @> '{NYC}' COLLATE "C"`**`;`
- `SELECT COUNT(*) FROM bookmarks WHERE` **`bookmark_tags_jsonb(js) @> '"NYC"'`**`;`
- `SELECT COUNT(*) FROM bookmarks WHERE` **`bookmark_tags_text(js) @> '{NYC}' COLLATE "C"`**`;`

Results are the same because only 1 entry is scanned:

```
0.812 ms jsonb_path_ops
0.787 ms jsonb_ops

0.829 ms jsonb_path_ops
0.793 ms jsonb_ops

0.176 ms (no recheck)

3.771 ms (expensive
recheck)

0.175 ms (no recheck)
```

Opclass parameters: jsonb results

- Index size and build time of selective indices are significaly lower than for the full ones, but they are same as for the functional indices on SQL/JSON functions or operators.
- Selective indexes allow more flexible queries
- Searches by functional **jsonb_ops** indices are faster than by full or selective ones because fewer index entries are scanned.
- Functional jsonb indices with subqueries in expressions are slow on both search and update.
- Searches by text[] indices are the fastest due to absence recheck.

Opclass parameters: jsonb .**

-
  ```
  CREATE TABLE js_test(id serial PRIMARY KEY, value jsonb);
  ```

-
  ```
  INSERT INTO js_test(value) VALUES
   ('[1, "a", true, {"b": "c", "f": false}]'),
   ('{"a": "blue", "t": [{"color": "red", "width": 100}]}'),
   ('[{"color": "red", "width": 100}]'),
   ('{"color": "red", "width": 100}'),
   ('{"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"}'),
   ('{"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}'),
   ('{"a": "blue", "t": [{"color": "blue", "width": 100}], "colr": "red"}'),
   ('{"a": "blue", "t": [{"color": "green", "width": 100}]}'),
   ('{"color": "green", "value": "red", "width": 100}');
  ```

-
  ```
  CREATE INDEX ON js_test USING gin
     (value jsonb_ops (projection="$.**.color"));
  ```

-
  ```
  SELECT id FROM js_test WHERE value @~ '$.**.color == "red"';
  ```

Opclass parameters: roadmap

GIN jsonb_ops and jsonb_path_ops:

- Filters for indexed paths:
  ```
  projection='$[*] ? (@ < 5)'
  ```
  Predicate implication proving is needed here.

- Expression indexing (like ordinary functional indices):
  ```
  projection='$.a + $.b'
  ```

- .datetime() support:
  ```
  projection='$.datetime()'
  js @~ '$.datetime() < "2018-02-06".datetime()'
  ```

- .type() support:
  ```
  js @~ '$.type() == "object"'
  ```

Opclass parameters: roadmap

- JsQuery GIN support:
  - indexed paths can be specified with new jsquery extension that allows us to extract fragments of jsonb:
    ```
    SELECT jsonb '[{"a":1,"b":2},{"a":3,"b":4}]' ~~ '#.a';
    [1, 3]
    ```

  -
    ```
    CREATE INDEX ON t USING gin
      (jb jsonb_value_path_ops(projection='#.a, b.c.#'));
    ```
  - JsQuery indexes can be queried with jsquery operator @@ or with jsonpath operators @? and @~:
    ```
    SELECT * FROM t WHERE jb @@ '#.a = 1 and b.c.# > 2';
    SELECT * FROM t WHERE jb @~ '$[*].a == 1 && $.b.c[*] > 2';
    ```

ALL
YOU
POSTGRES
NEED
IS