K-Nearest Neighbors Search in PostgreSQL

Oleg Bartunov, Nikita Glukhov
Postgres Professional

Research scientist @ Moscow University
CEO Postgres Professional
Major PostgreSQL contributor
Since Postgres95
Nikita Glukhov

Senior developer @Postgres Professional
PostgreSQL contributor

Major CORE contributions:
- Jsonb improvements
- SQL/JSON (Jsonpath)
- KNN SP-GiST
- Opclass parameters

Current development:
- SQL/JSON functions
- Jsonb performance
Knn-search: The problem

- What are interesting points near Royal Oak pub in Ottawa?
- What are the closest events to the May 20, 2009 in Ottawa?
- Similar images – feature extraction, Hamming distance
- Classification problem (major voting)
- ............
- GIS, Science (high-dimensional data)
K-nearest neighbour search

10 closest events to the launch of Sputnik?

```
SELECT id, date, event FROM events ORDER BY date <-> '1957-10-04'::date ASC LIMIT 10;
```

<table>
<thead>
<tr>
<th>id</th>
<th>date</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>58136</td>
<td>1957-10-04</td>
<td>&quot;Leave It to Beaver,&quot; debuts on CBS</td>
</tr>
<tr>
<td>58137</td>
<td>1957-10-04</td>
<td>U.S.S.R. launches Sputnik I, 1st artificial Earth satellite</td>
</tr>
<tr>
<td>117062</td>
<td>1957-10-04</td>
<td>Gregory T Linteris, Demarest, New Jersey, astronaut, sk: STS 83</td>
</tr>
<tr>
<td>117061</td>
<td>1957-10-04</td>
<td>Christina Smith, born in Miami, Florida, playmate, Mar, 1978</td>
</tr>
<tr>
<td>102671</td>
<td>1957-10-05</td>
<td>Lee &quot;Kix&quot; Thompson, saxophonist, Madness-Baggy Trousers</td>
</tr>
<tr>
<td>102670</td>
<td>1957-10-05</td>
<td>Larry Saumell, jockey</td>
</tr>
<tr>
<td>58292</td>
<td>1957-10-05</td>
<td>Yugoslav dissident Milovan Djilos sentenced to 7 years</td>
</tr>
<tr>
<td>58290</td>
<td>1957-10-05</td>
<td>11th NHL All-Star Game: All-Stars beat Montreal 5-3 at Montreal</td>
</tr>
<tr>
<td>31456</td>
<td>1957-10-03</td>
<td>Willy Brandt elected mayor of West Berlin</td>
</tr>
<tr>
<td>58291</td>
<td>1957-10-05</td>
<td>12th Ryder Cup: Britain-Ireland, 7 - 4 at Lindrick GC, England</td>
</tr>
</tbody>
</table>

Before 9.1: Slow: Index is useless (no WHERE!), full heap scan, sort, limit

Limit (actual time=54.481..54.485 rows=10 loops=1)
Buffers: shared hit=1824
  -> Sort (actual time=54.479..54.481 rows=10 loops=1)
    Sort Key: (abs((date - '1957-10-04'::date)))
    Sort Method: top-N heapsort Memory: 26kB
    Buffers: shared hit=1824
  -> Seq Scan on events (actual time=0.020..25.896 rows=151643 loops=1)
    Buffers: shared hit=1824
Planning Time: 0.091 ms
Execution Time: 54.513 ms
(10 rows)
K-nearest neighbour search

• 10 closest events to the launch of Sputnik
• Before 9.1: Slow: Index is useless, full heap scan, sort, limit

```
Limit (actual time=54.481..54.485 rows=10 loops=1)
  Buffers: shared hit=1824
  ->  Sort (actual time=54.479..54.481 rows=10 loops=1)
      Sort Key: (abs((date - '1957-10-04'::date)))
      Sort Method: top-N heapsort  Memory: 26kB
      Buffers: shared hit=1824
      ->  Seq Scan on events (actual time=0.020..25.896 rows=151643 loops=1)
          Buffers: shared hit=1824
Planning Time: 0.091 ms
Execution Time: 54.513 ms
(10 rows)
```

• After 9.1: Amazingly FAST using index!

```
Limit (actual time=0.123..0.143 rows=10 loops=1)
  Buffers: shared hit=9
  ->  Index Scan using event_date_idx on events (actual time=0.122..0.138 rows=10 loops=1)
      Order By: (date <-> '1957-10-04'::date)
      Buffers: shared hit=9
Planning Time: 0.105 ms
Execution Time: 0.172 ms
(7 rows)
```

Index can be used for
• Filtering - WHERE expr opr value
• Ordering - ORDER BY expr [ASC|DESC]
• K-NN   - ORDER BY expr opr value [ASC]

~ 300 times faster!
Knn-search: Existing solutions

Traditional way to speedup such query

• Use indexes - very inefficient (no search query !)
  • Scan full index
  • Full table scan, but in random order !
  • Sort full table
  • Better not to use index at all !

• Constrain data space (range search) - "Костыль"
  • Incremental search → to many queries
  • Need to know in advance size of neighbourhood, how ?
    1Km is ok for Paris, but too small for Siberia
  • Maintain 'density map' ?
Difficult choice for initial radius
Regular Index traverse

• Index is a search tree

Depth First Search
(stack, LIFO)

Breadth First Search
(queue, FIFO)

Both strategies require WHERE clause, for K-NN it means full index scan
K-NN Index traverse — Best First Search

• Best First Search (PQ, priority queue). Maintain order of items in PQ according their distance from given point
  - Distance to MBR (rectangle for Rtree) for internal pages – minimum distance of all items in that MBR
  - Distance = 0 for MBR with given point
  - Distance to point for leaf pages

• Each time we extract point from PQ we output it – it is the next closest point! If we extract rectangle, we expand it by pushing their children (rectangles and points) into the queue.

• We traverse index by visiting only interesting nodes!
K-NN Index traverse — Best First Search

- Simple example – non-overlapped partitioning
K-NN Index traverse — Best First Search

- Simple example – non-overlapped partitioning

- Priority Queue
  - 1: \{1, 2, 3, 4, 5, 6, 7, 8, 9\}
  - 2: \{2, 5, 6, 7, 9\}, \{1, 3, 4, 8\}
  - 3: \{5, 6, 7, 9\}, \{1, 3, 4, 8\}, \{2\}
  - 4: \{5, 9\}, \{1, 3, 4, 8\}, \{2\}, \{6, 7\}
  - 5: \{1, 3, 4, 8\}, 5, \{2\}, \{6, 7\}, 9
  - 6: \{1, 3, 4\}, \{8\}, 5, \{2\}, \{6, 7\}, 9
  - 7: 4, \{8\}, 5, \{2\}, \{6, 7\}, 3, 1, 9
  - 8: 5, \{2\}, \{6, 7\}, 3, 8, 1, 9
  - 9: \{6, 7\}, 3, 2, 8, 1, 9
  - 10: 3, 2, 8, 1, 9, 6, 7

we can output 4 without visit other rectangles!
K-NN Index traverse — Best First Search

- Simple example – non-overlapped partitioning

- Priority Queue

  - 1: \{1, 2, 3, 4, 5, 6, 7, 8, 9\}
  - 2: \{2, 5, 6, 7, 9\}, \{1, 3, 4, 8\}
  - 3: \{5, 6, 7, 9\}, \{1, 3, 4, 8\}, \{2\}
  - 4: \{5, 9\}, \{1, 3, 4, 8\}, \{2\}, \{6, 7\}
  - 5: \{1, 3, 4, 8\}, 5, \{2\}, \{6, 7\}, 9
  - 6: \{1, 3, 4\}, \{8\}, 5, \{2\}, \{6, 7\}, 9
  - 7: 4, \{8\}, 5, \{2\}, \{6, 7\}, 3, 1, 9

We can output 4 without visit other rectangles!

- 8: 5, \{2\}, \{6, 7\}, 3, 8, 1, 9
- 9: \{6, 7\}, 3, 2, 8, 1, 9
- 10: 3, 2, 8, 1, 9, 6, 7
Knn-search: What we did!

- + We want to avoid full table scan – read only <right> tuples
  - So, we need index

- + We want to avoid sorting – read <right> tuples in <right> order
  - So, we need special strategy to traverse index

- + We want to support tuples visibility
  - So, we should be able to resume index traverse

- + We want to support many data types
  - So, we need to modify GiST
GiST Modification

**Depth First Search**

push Stack, Root;  
While Stack {
   If p is heap {
      output p;
   } else {
      children = get_children(p);
      push Stack, children;
   }
}

**Best First Search**

push PQ, Root;  
While PQ {
   If p is heap {
      output p;
   } else {
      Children = get_children(p);
      push PQ, children;
   }
}

- For non-knn search all distances are zero, so PQ => Stack and BFS => DFS
- We can use only one strategy for both – normal search and knn-search!
Knn-search: Performance

- SEQ (no index) – base performance
  - Sequentially read full table + Sort full table (can be very bad, sort_mem !)

- DFS — very bas !
  - Full index scan + Random read full table + Sort full table

- BFS – the best for small k !
  - Partial index scan + Random read k-records
    - \( T(\text{index scan}) \sim \text{Height of Search tree} \sim \log(n) \)
  - Performance win BFS/SEQ \sim \text{Nrelpages/k}, for small k.
    - The more rows, the more benefit !
  - Can still win even for \( k=n \) (for large tables) - no sort !
Example: GiST R-Tree for Greece

• Create two tables with a subset of geo_all data belonging to Greece:
  • CREATE TABLE geo_greece AS SELECT * FROM geo_all WHERE country_code = 'GR';
  • CREATE TABLE geo_greece_rnd AS SELECT * FROM geo_greece ORDER BY random();

• Create three GiST indexes:
  • Z-order sorted which is default for PG14 (WITH (buffering = off) is by default)
    CREATE INDEX geo_greece_zorder_idx ON geo_greece USING gist (location);
  • Buffered on unsorted data
    CREATE INDEX geo_greece_unsorted_idx ON geo_greece USING gist(location) WITH (buffering=on);
  • Buffered on randomized data
    CREATE INDEX geo_greece_rnd_idx  ON geo_greece_rnd USING gist(location) WITH (buffering=on);
Example: GiST R-Tree for Greece

• Index statistics:

<table>
<thead>
<tr>
<th>index</th>
<th>size</th>
<th>build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>geo_greece_rnd_idx</td>
<td>2632 kB</td>
<td>220 ms</td>
</tr>
<tr>
<td>geo_greece_unsorted_idx</td>
<td>2736 kB</td>
<td>240 ms</td>
</tr>
<tr>
<td>geo_greece_zorder_idx</td>
<td>1768 kB</td>
<td>37 ms</td>
</tr>
</tbody>
</table>

• R-Tree levels statistics (Gevel extension):

```
SELECT relname, level, count(*)
FROM pg_class, gist_print(relname) AS t(level int, valid bool, a box)
WHERE relname LIKE 'geo_greece_%_idx'
GROUP BY 1, 2 ORDER BY 1, 2;
```

<table>
<thead>
<tr>
<th>relname</th>
<th>level</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>geo_greece_rnd_idx</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>geo_greece_rnd_idx</td>
<td>2</td>
<td>324</td>
</tr>
<tr>
<td>geo_greece_rnd_idx</td>
<td>3</td>
<td>36124</td>
</tr>
<tr>
<td>geo_greece_unsorted_idx</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>geo_greece_unsorted_idx</td>
<td>2</td>
<td>335</td>
</tr>
<tr>
<td>geo_greece_unsorted_idx</td>
<td>3</td>
<td>36124</td>
</tr>
<tr>
<td>geo_greece_zorder_idx</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>geo_greece_zorder_idx</td>
<td>2</td>
<td>218</td>
</tr>
<tr>
<td>geo_greece_zorder_idx</td>
<td>3</td>
<td>36124</td>
</tr>
</tbody>
</table>

1.5x more leaves per internal node
Example: GiST R-Tree for Greece

Z-order index boxes overlaps more strongly
Example: GiST R-Tree for Greece – search performance

Search speed depending on location and search area.
SELECT count(*) FROM geo_greece WHERE location <@ circle(point(lng, lat), radius);
Example: GiST R-Tree for Greece — search performance

Amount of blocks read depending on location and search area.

SELECT count(*) FROM geo_greece WHERE location <@ circle(point(lng, lat), radius);
Example: GiST R-Tree for Greece — search performance

Search speed for small areas is determined by node overlapping (radius = 0.01):
Example: GiST R-Tree for Greece – kNN performance

kNN search time is not much tied to tree partitioning, but kNN still suffers from overlapping.

```
SELECT count(*) FROM geo_greece ORDER BY location <-> point(lng, lat) LIMIT n;
```
Example: GiST R-Tree for Greece – kNN performance

Number of index blocks read by lower-limit kNN is greatly increased by tree overlapping.

```
SELECT count(*) FROM geo_greece ORDER BY location <-> point(lng, lat) LIMIT n;
```
Whole world kNN – Geonames dataset

- Geonames dataset contains coordinates of 422523 cities
McMurdo Station, Antarctic research center — the farthest settlement in geonames db

<table>
<thead>
<tr>
<th>geonameid</th>
<th>earth_location</th>
<th>dist (postgis)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6696480</td>
<td>McMurdo Station</td>
<td>(166.676,-77.846)</td>
</tr>
</tbody>
</table>
KNN whole world – DIY kNN

- KNN can be emulated by searching and sorting points inside the circles with exponentially increasing radius. Its performance is quite sensitive to the choice of initial radius, which can be estimated by point density and search limit.

```sql
SELECT points.*
FROM
    (VALUES (point($2, $1))) p(pt),
    generate_series(0.0, 20.0, 1.0) i,
    pow(2.0, i + $4) r,      -- increasing circle radius, starting from 2^$4
LATERAL (
    SELECT * FROM (  
        SELECT geonameid, name, location, location <-> pt dist  
        FROM cities  
        WHERE location @@ circle(pt, r)        -- search point inside circle
    ) tmp
    WHERE i = 0 OR dist > r * 0.5        -- skip points inside previous circle
    ORDER BY dist
) points
LIMIT $3;
```
• GiST kNN is faster than SP-GiST kNN in lower-density areas
• Seq scan is a bit faster than kNN by index only when all rows are read
• DIY kNN scan speed is very sensitive to initial radius for low limits
KNN whole world – Europe vs Siberia

- The same applies to the number of index blocks read.
Whole world kNN – geonames dataset

- Geonames dataset contains coordinates of 422523 cities
Whole world kNN – distance functions

- Built-in opclass GiST and SP-GiST opclasses for points use simple Euclid 2D distance: \( \sqrt{(\text{lat}_1 - \text{lat}_2)^2 + (\text{lng}_1 - \text{lng}_2)^2} \)
- PostGIS uses correct spherical distance function like contrib/earthdistance.
- The difference between them is absent for near points on equator, but it grows up with distance and latitude growth.
Whole world kNN – GiST Z-order sorted index

Z-order sorting index produces a lot of overlapping.
Whole world kNN – GiST on randomly sorted data

Buffered GiST building with randomly sorted data produces a much better tree partitioning.
Whole world kNN – SP-GiST Quad-Tree index

SP-GiST Quad-Tree contains up to 19 levels in higher-density areas.
Whole world kNN – SP-GiST KD-Tree index

SP-GiST KD-Tree has even more levels: up to 32 levels.
Whole world kNN – 4 indexes, Europe only

GiST randomized
GiST Z-order
SP-GiST Quad
SP-GiST KD
KNN whole world – execution time map
KNN whole world – index blocks map
KNN whole world – total blocks map
KNN whole world – Z-order sorted vs randomized

- Z-order performance is more unstable
KNN whole world – 4 indexes

- GiST rnd has the best predictable performance
KNN on trigrams

• contrib/pg_trgm
• GiST and GIN indexes
• Supports kNN only on GiST
• Distance operators:
  • Similarity: text <-> text
  • Word similarity: text <-> text, text <--> text

\[
\text{word1} \leftrightarrow \text{word2} = \frac{1 - N_{\text{common trigrams}}}{(\text{Len}_{\text{word1}} + \text{Len}_{\text{word2}} - N_{\text{common trigrams}})}
\]
KNN on trigrams

pg_gtrm GiST index contains:

- Unmodified arrays of trigrams in the leaves.
- Fixed-length signatures in the inner nodes. Each trigram is mapped to the one bit in signature using simple hash function. So trigrams can collide and signature lengths equal to power of 2 are not effective, because hash function will be not sensitive to all chars.

\[
\text{Hash}(\text{trg}) = (\text{char}_0 + \text{char}_1 \ll 8 + \text{char}_2 \ll 16) \mod \text{len}_{\text{signature}}
\]

\[
\text{Signature}(\text{word}) = \bigvee_{i=1..N_{\text{trg}}} (1 \ll \text{Hash}(\text{trg}_i))
\]

\[
\text{Signature}(\text{leaf page}) = \bigvee_{i=1..N_{\text{words}}} (\text{Signature}(\text{word}_i))
\]

\[
\text{Signature}(\text{inner page}) = \bigvee_{i=1..N_{\text{items}}} (\text{Signature}_i)
\]

- kNN uses the following minimal distance estimation for signatures:

\[
\text{word <-> signature} = 1 - \frac{N_{\text{common bits}}}{N_{\text{word trigrams}}}
\]
KNN on trigrams

Sample GiST tree with 8 words and query:
SELECT * FROM words ORDER BY word <-> 'georges' LIMIT 3;
KNN on trigrams

Step 1: calculate distance to root items
KNN on trigrams

Step 2: descent to children of nearest root item, other items are queued
KNN on trigrams

Step 3: descent to 2nd leaf page of the nearest inner item
KNN on trigrams

Step 4: emit first leaf item because it is the nearest in the queue
Step 5: emit second leaf item, it is also the nearest in the queue
KNN on trigrams

Step 6: descent to the remaining inner node
KNN on trigrams

Step 7: descent to the 3rd leaf page
KNN on trigrams

Step 8: emit the second leaf item of 3rd leaf page

1: \{R1,R2\}
2: \{I2,R2,I1\}
3: \{L3,L4,R2,I1\}
4: \{L3\}\{L4,R2,I1\}
5: \{L4\}\{R2,I1\}
6: \{I3,I4,I1\}
7: \{L6,I4,L5,I1\}
8: \{L6\}\{I4,L5,I1\}
KNN on trigrams

Results (average time, 100 random sample words)
Search Similar images - OK
Search Similar images - OK
Search Similar images — NOT OK
CREATE TABLE pat AS (
  SELECT
    id,
    shuffle_pattern(pattern) AS pattern,
    pattern2signature(pattern) AS signature
  FROM (SELECT
    id,
    jpeg2pattern(data) AS pattern
  FROM image
  ) x
);
CREATE INDEX pat_signature_idx ON pat USING gist (signature);
CREATE INDEX pat_id_idx ON pat(id);
SELECT id,
       smlr
FROM (SELECT id,
            pattern <-> (SELECT pattern FROM pat WHERE id = :id) AS smlr
       FROM pat
       WHERE id <> :id
       ORDER BY signature <-> (SELECT signature FROM pat WHERE id = :id)
       LIMIT 100
      ) x
ORDER BY x.smlr ASC
LIMIT 10
Search similar: Inside

B&W and resize → jpeg2pattern → Haar wavelet → shuffle pattern → pattern2signature
K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q!

CREATE table circle (id serial, p point, s int4);
INSEART INTO circle (p,s)
    SELECT point( p.x, p.y), (random()*1000)::int
    FROM ( select t.x, sqrt(1- t.x*t.x) AS y
            FROM ( select random() as x, generate_series(1,1000000) ) as t
    ) AS p;
CREATE index circle_p_idx ON circle USING gist(p);
ANALYZING circle;

Number of levels: 3
Number of pages: 8266
Number of leaf pages: 8201
K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q!

```sql
explain (analyze on, buffers, costs off) select * from circle order by (p <-> '(0,0)') asc limit 10;
```

**QUERY PLAN**

<table>
<thead>
<tr>
<th>Limit (actual time=252.242..252.244 rows=10 loops=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers: shared hit=7353</td>
</tr>
<tr>
<td>-&gt; Sort (actual time=252.240..252.241 rows=10 loops=1)</td>
</tr>
<tr>
<td>Sort Key: ((p &lt;-&gt; '0,0')::point))</td>
</tr>
<tr>
<td>Sort Method: top-N heapsort Memory: 26kB</td>
</tr>
<tr>
<td>Buffers: shared hit=7353</td>
</tr>
<tr>
<td>-&gt; Seq Scan on circle (actual time=0.015..140.504 rows=1000000 loops=1)</td>
</tr>
<tr>
<td>Buffers: shared hit=7353</td>
</tr>
</tbody>
</table>

Planning Time: 0.103 ms
Execution Time: **252.272** ms
(10 rows)
K-NN Corner Case

• Corner case for Best First Strategy - all data are on the same distance from point Q!

explain (analyze on, buffers, costs off) select * from circle order by (p <-> '(0,0)') asc limit 10;

QUERY PLAN

----------------------------------------------------------------------------------
| Limit (actual time=96.037..96.057 rows=10 loops=1) |
| Buffers: shared hit=6073 |
| -> Index Scan using circle_p_idx on circle (actual time=96.035..96.054 rows=10 loops=1) |
| Order By: (p <-> '(0,0) '::point) |
| Buffers: shared hit=6073 |
Planning Time: 0.065 ms
Execution Time: **125.737** ms VS **252.272** ms (seq scan)
(7 rows)
K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q!

```
Limit (actual time=97.266..99.453 rows=10 loops=1)
  Buffers: shared hit=7431
  -> Gather Merge (actual time=97.265..99.449 rows=10 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    Buffers: shared hit=7431
    -> Sort (actual time=92.849..92.850 rows=10 loops=3)
      Sort Key: ((p <-> '(0,0)'::point))
      Sort Method: top-N heapsort  Memory: 26kB
      Buffers: shared hit=7431
      Worker 0: Sort Method: top-N heapsort  Memory: 26kB
      Worker 1: Sort Method: top-N heapsort  Memory: 25kB
      -> Parallel Seq Scan on circle (actual time=0.013..53.711 rows=333333 loops=3)
        Buffers: shared hit=7353
```
Planning Time: 0.076 ms
Execution Time: 99.477 ms  (seq scan parallel) < **125.737 (knn)** ms VS **252.272 ms** (seq scan) (16 rows)
TODO

• Modify opclass API to support different distance operators independently, without need to rewrite support function shared between them

• Support for reverse distance ordering, NULLS FIRST:
  ORDER BY distance DESC [NULLS FIRST]
  ORDER BY distance ASC NULLS FIRST

• Replace sortsupport opclass function with opclass parameter:
  CREATE INDEX ... USING (col1 opclass1 (sort=zorder_point_cmp), ...)

• Investigate hot spots on kNN map

• Try Z-order sorting for Quad-Tree SP-GiST indexes building
TODO – multiple distance operators support

• Currently, GiST needs support function to opclass, that will handle all possible kNN strategies. So, you can’t easily add new ordering operator to opclass.

CREATE OPERATOR CLASS foo_ops …
  OPERATOR 1 <-> FOR ORDER BY bar_pos,
  OPERATOR 2 <@> FOR ORDER BY baz_ops,
  FUNCTION 8 gist_foo_distance … -- should handle both operators

• The possible solution is support functions for strategies:
  FUNCTION 8 gist_foo_bar_distance FOR STRATEGY 1,
  FUNCTION 8 gist_foo_baz_distance FOR STRATEGY 2,

• The same applies to consistent function, used for search operators.

• In SP-GiST it is even worse: kNN support is hardcoded into consistent support function. At fist, we need to extract distance support function.
References

- Original K-NN post in hackers, we introduced \(<\) operator, now it is \(<-\)
  - https://www.postgresql.org/message-id/4B0AC9E1.5050509@sigaev.ru
- Original K-NN talk at PGCON-2010
- Slides of this talk (PDF)
- Geonames database dump
  - http://download.geonames.org/export/dump/
- Events database dump
  - http://www.sai.msu.su/~megera/postgres/files/events.dump.gz
- Gevel extension – a tool for inspecting indexes
  - git://sigaev.ru/gevel