Index support for regular expression search

Alexander Korotkov
Introduction
What are regular expressions?

Regular expressions are:

• powerful tool for text processing
• based on formal language theory
• expressing same class of “languages” as finite automata
Automata... didn't hear about it
So... automata

• Regular expression can be transformed into automaton.
• Moreover, such transformation is really used by regex engines
So... automata

- Automaton is a graph which vertices are “states” and which arcs are labeled by characters.
- Automaton “reads” string if you can type that string by a traversal from “initial” state to “final” state.
Example

\(/a(bc)^*d/\)
becomes
Example

xyzabcdxyz
Example

\texttt{xyzabcbcdxyz}
Example

xyzabcabcdxyz
Example

xyzabcbcdxyz
Example

$xyzabcbcdxyz$
Example

xyzabcbcdxyz
Example

xyzabcbcdxyz
Example

xyzabcbcdxyz
Example

xyzabcbcdxyz
Example

xyzabcabcdxyz

[Diagram of a state machine with transitions labeled a, b, c, and d, and states marked with asterisk, red circle, and yellow circle.]
Example

xyzabcabcdxyz
Example

xyzabcabcdxyz
Example

xyzabcabcdxyz

Finish! Match!

Diagram: A non-deterministic finite automaton (NFA) for the pattern matching example.
Okay, now all of us know...
Regex based search

- PostgreSQL can regex based search :)
- It’s only a sequential search for a while :(
Inverted indexes on q-grams
Q-grams?
Q-grams

- Q-gram is substring of length q which can be used as a signature of original string
- Widely used in various string processing tasks
Inverted index on q-grams

- Maintain association between q-gram and all the strings where it mentioned.
- `pg_trgm` has an implementation for \( q = 3 \)
1. "regular expressions",
2. "expressive speech",
3. "regular speaker"

=>

<table>
<thead>
<tr>
<th>Character</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'e'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'egu'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'pre'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'r'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'er'</td>
<td>{3}</td>
</tr>
<tr>
<td>'reg'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'s'</td>
<td>{2,3}</td>
</tr>
<tr>
<td>'ess'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'res'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'ex'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'exp'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'sio'</td>
<td>{1}</td>
</tr>
<tr>
<td>'re'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'gul'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'siv'</td>
<td>{2}</td>
</tr>
<tr>
<td>'sp'</td>
<td>{2,3}</td>
</tr>
<tr>
<td>'siv'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ion'</td>
<td>{1}</td>
</tr>
<tr>
<td>'spe'</td>
<td>{2,3}</td>
</tr>
<tr>
<td>'ach'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ive'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ssi'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'ake'</td>
<td>{3}</td>
</tr>
<tr>
<td>'ker'</td>
<td>{3}</td>
</tr>
<tr>
<td>'ula'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'ar'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'lar'</td>
<td>{1,3}</td>
</tr>
<tr>
<td>'ve'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ch'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ns'</td>
<td>{1}</td>
</tr>
<tr>
<td>'xpr'</td>
<td>{1,2}</td>
</tr>
<tr>
<td>'eac'</td>
<td>{2}</td>
</tr>
<tr>
<td>'ons'</td>
<td>{1}</td>
</tr>
<tr>
<td>'eak'</td>
<td>{3}</td>
</tr>
<tr>
<td>'pea'</td>
<td>{2,3}</td>
</tr>
</tbody>
</table>
Q-grams frequencies

From 2.5M of DBLP paper titles

- 360K contain trigram “the”
- Only 1 contains trigram “zzz”

"Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz" chapter of the book "Formal Specification and Development in Z and B" by David Everett

Not all q-grams of fixed q are equally useful.
Q-grams are not equally useful

I was running a factor analysis on this huge database, and check out what it found:

Sexual arousal

Consecutive vowels

Huh? This chart makes no sense. What—"Queueing"

Fuck me now.
V-grams or multigrams

- Each q-gram can have specific q
- Selectivity of all q-grams are similar (and low enough)
- More effective index search!
V-grams or multigrams

Problems:

• Hard to maintain online

• …
Patent trololo!
How to use it for regex search?
General idea

/\[ab\]cde/ => (acd OR bcd) AND cde
acd: \{1,4,5\}, bcd: \{2,3,4\}, cde: \{2,4,6\}

\begin{array}{cccc}
acd & bcd & cde & (acd OR bcd) AND cde \\
\hline
1 & t & f & f \\
2 & f & t & t \\
3 & f & t & f \\
4 & t & t & t \\
5 & t & f & f \\
6 & f & f & t \\
\end{array}

Index support for regular expression search
PGConf.EU-2012, Prague
General idea

/\[ab\]cde/ => (acd OR bcd) AND cde

How to do this in general case?
Existing approaches for q-gram extraction
Scholar paper

Junghoo Ch and Sridhar Rajagopalan, *A fast regular expression indexing engine*, Proceedings 18th International Conference on Data Engineering, 2002

Still widely referenced as state of art work about indexing for regular expressions.
FREE method

• Extract tree of continuous string fraction from regex.
• Transform those continuous fractions to multigrams (q-grams with variable q).
• Use inverted index on multigrams for query evaluation
FREE method: example

Tree for /\(abcd|efgh)(ijklm|x*)/\n
```
  AND
 (    )
(OR) OR
  (    )
  (    )
   OR
    (    )
     (    )
      (abcd)
      (efgh)
      (ijklm)
      (x)
```
Replace "*" nodes with NULL
NULL “eats” parent OR node

\[
\text{AND} \quad \text{OR} \quad \text{null}
\]

\[
\text{abcd} \quad \text{efgh}
\]
AND node “eats” child NULL

AND

OR

abcd  efgh
Simplify a bit

OR

abcd  efgh
Expand continuous string fractions into trigrams

\[
\begin{align*}
\text{OR} & \quad \text{AND} \quad \text{AND} \\
\quad abc & \quad bcd & \quad efg & \quad fgh
\end{align*}
\]
Google code search

• Was launched in 2006.
• Supports regex search.
• Google guys are smart. It can’t be a sequential scan.
• It also seemed to be something better than previous technique.
• We don’t know what... :(
We didn’t know what until...

- Google code search was closed in 2011 :(
- Russ Cox has published description of indexing technique in January 2012

http://swtch.com/~rsc/regexp/regexp4.html

- More than 5 years of intrigue!
Google code search method

• Get 5 characteristics about each part of regex: emptyable, exact, prefix, suffix, match.

• Recursively union them (with possible simplification)

• Use inverted index of trigrams for query evaluation (similar to pg_trgm)
Google code search method

Original regex: /a(bc)+d/
a: {exact: a}
bc: {exact: bc}
d: {exact: d}
(bc)+: {prefix: bc, suffix: bc}
a(bc)+: {prefix: abc, suffix: bc}
a(bc)+d: {prefix: abc, suffix: bcd}
Google code search method

\[ /a(bc)+d/ \]

\{prefix:abc, suffix:bcd\}

abc AND bcd
Proposed method
Proposed method

- Transform automaton into automaton like graph on trigrams
- Simplify that graph
- Use pg_trgm indexes
Transformation example

/a(b+|c+)d/
Transformation example

![Diagram of a transformation example with nodes and edges labeled with characters and arrows indicating transitions.](image)
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example

Diagram:

- Nodes: 1, 2, 3, 4, 5
- Edges:
  - 1 → 2 → 4 → 5
  - 1 → 3
  - 2 → 4
  - 3 → 4

Labels:
- Node 1: a
- Node 2: b
- Node 3: c
- Node 4: A, b
- Node 5: #

Transactions:
- (ab, 2)
- (ac, 3)
- (ac, 4)
- (bb, 2)
- (bb, 4)

Abbreviation: abb
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example
Transformation example

Result could be simplified
Transformation example

Implemented simplification technique:

collect path matrix.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Means: abd OR (abb AND bbd) OR acd OR (acc AND ccd)
Comparison on examples

**Regex:** /(abc|cba)def/

**FREE:** (abc OR cba) AND def

**GSC:**
def AND ((abc AND bcd AND cde) OR (ade AND bad AND cba))

**My:**
(abc AND bcd AND cde AND def) OR (ade AND bad AND cba AND def)
Comparison on examples

Regex: /abc+de/

FREE: nothing

GSC: abc AND cde

My:

(abc AND cde AND bcd) OR
(abc AND cde AND bcc AND ccd)
Comparison on examples

Regex: /(abc*)+de/

FREE: nothing

GSC: nothing

My:
(abd AND bde) OR
(abc AND bcd AND cde) OR
(abc AND bcc AND ccd AND cde)
Comparison on examples

Regex: /ab(cd)*ef/

FREE: nothing

GSC: nothing

My:

(abe AND bef) OR

(abc AND bde AND cde AND def)
## Performance results

2.5 M DBLP paper titles of 47 avg. length

<table>
<thead>
<tr>
<th>Regex</th>
<th>Index scan</th>
<th>Seq scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>`/database.*(sql</td>
<td>query)/`</td>
<td>773 ms</td>
</tr>
<tr>
<td><code>/postgres(ql)?/</code></td>
<td>268 ms</td>
<td>17574 ms</td>
</tr>
<tr>
<td><code>/plan+er/</code></td>
<td>253 ms</td>
<td>12885 ms</td>
</tr>
<tr>
<td>`/(nucl</td>
<td>anino).*acid/`</td>
<td>200 ms</td>
</tr>
<tr>
<td><code>/[aei](bc)+a/</code></td>
<td>2 ms</td>
<td>13195 ms</td>
</tr>
</tbody>
</table>
Help needed

Regular expressions and string collections from real-life tasks for proving effectiveness of proposed method.
New since PGCon 2012

- Have an optimization for making resulting graph smaller.
- Can use both graph and path matrix (which is simpler)
- Encoding problem is solved.

New patch will be posted soon.
GIN improvements
Summary of changes

- Compressed storage with additional information
- Optimized search («frequent_entry & rare_entry» case)
- Return ordered results by index (ORDER BY optimization)

interface changes needs for all this stuff
Every GIN application can have a benefit

- Fulltext search: store word positions, get results in relevance order.
- Trigram indexes: store trigram positions, get results in similarity order.
- Array indexes: store array length, get results in similarity order.
Store additional information

Use increments and variable byte encoding to keep index small

1034, 1036, 1038 (12 bytes) => 1034, 2, 2 (4 bytes)
Fast scan

entry1 && entry2

Visiting 3 pages instead of 7
ORDER BY using index

Before

SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY ts_rank(fts, to_tsquery('english', 'query')) DESC
LIMIT 10;

After

SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY fts ~> to_tsquery('english', 'query')
LIMIT 10;

Ranking and sorting are outside the fulltext index

Index returns data ordered by rank. Ranking and sorting are inside.

368 ms vs 13 ms
Example: frequent entry (30%)

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Heap Scan</td>
<td>1</td>
<td>367.687 ms</td>
<td>94.6 %</td>
</tr>
<tr>
<td>Bitmap Index Scan</td>
<td>1</td>
<td>6.570 ms</td>
<td>1.7 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Sort</td>
<td>1</td>
<td>14.465 ms</td>
<td>3.7 %</td>
</tr>
</tbody>
</table>

Before:

388 ms

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Scan</td>
<td>1</td>
<td>13.346 ms</td>
<td>100.0 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>0.0 %</td>
</tr>
</tbody>
</table>

After:

13 ms
Example: rare entry (0.08%)
Example: frequent entry (30%) & rare entry (0.08%)

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Heap Scan</td>
<td>1</td>
<td>1.547 ms</td>
<td>23.0 %</td>
</tr>
<tr>
<td>Bitmap Index Scan</td>
<td>1</td>
<td>5.151 ms</td>
<td>76.7 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.000 ms</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Sort</td>
<td>1</td>
<td>0.022 ms</td>
<td>0.3 %</td>
</tr>
</tbody>
</table>

Before:

6.7 ms

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Scan</td>
<td>1</td>
<td>0.998 ms</td>
<td>100.0 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.000 ms</td>
<td>0.0 %</td>
</tr>
</tbody>
</table>

After:

1.0 ms
Thank you for attention!
Sponsors are welcome!