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

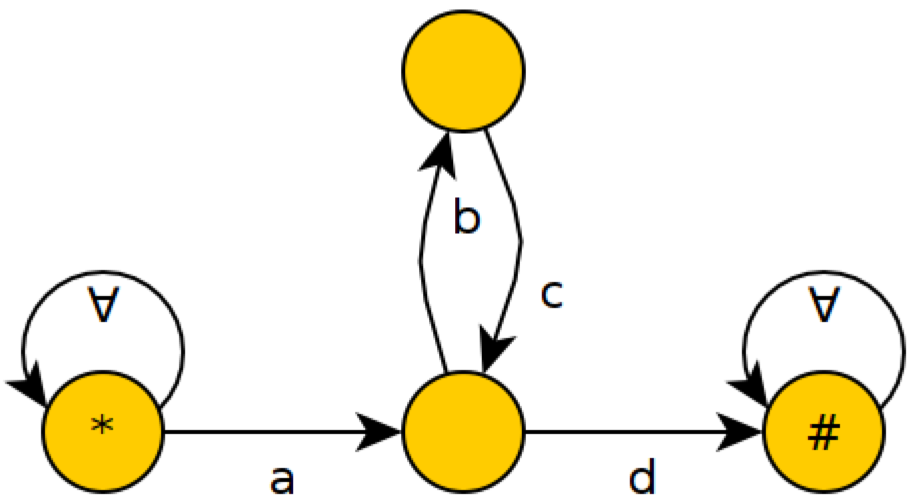
So... automata

- 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

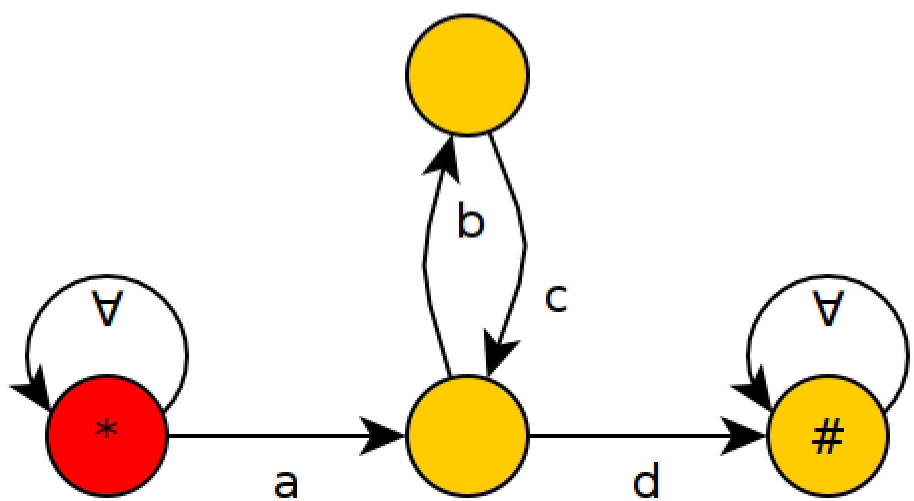
/a(bc)*d/

becomes



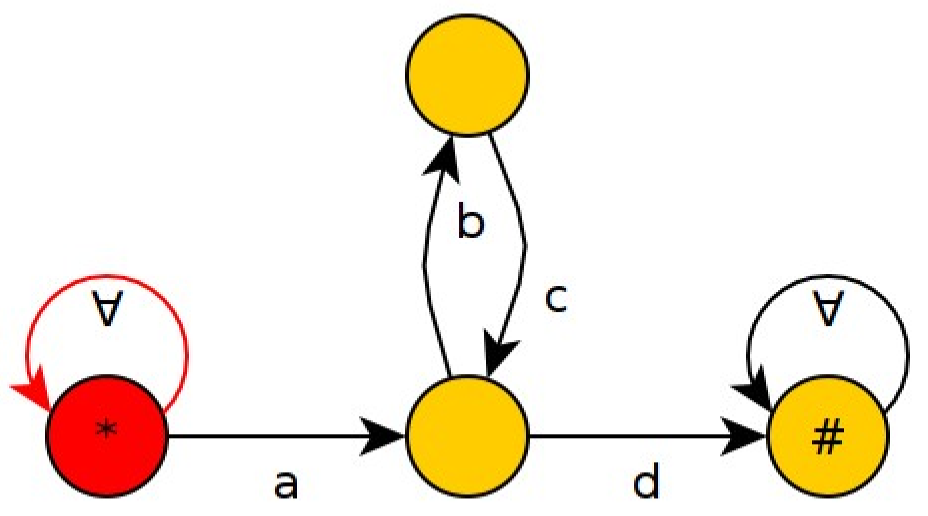
Example

xyzabcbcdxyz



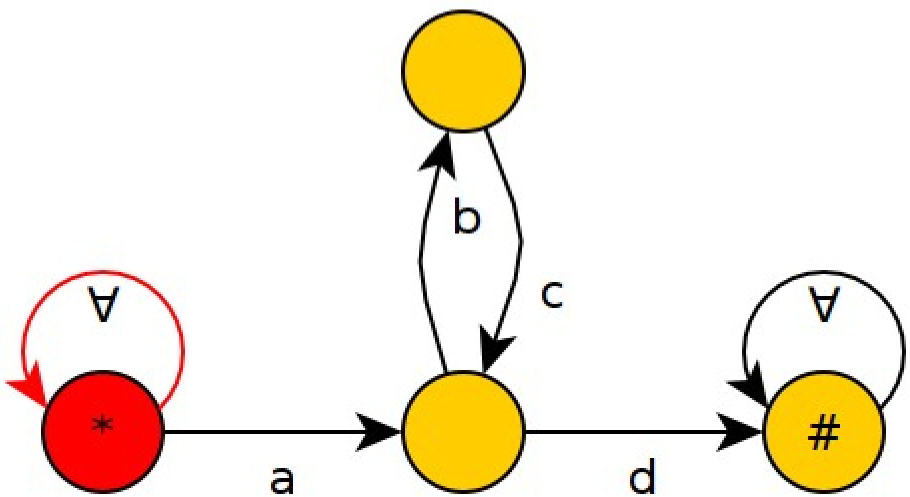
Example

xyzabcbcdxyz



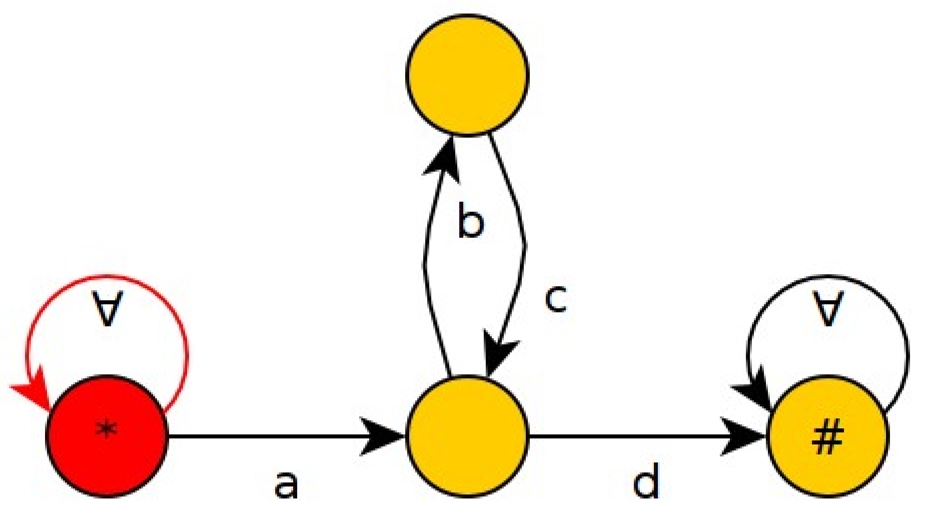
Example

xyzabcbcdxyz



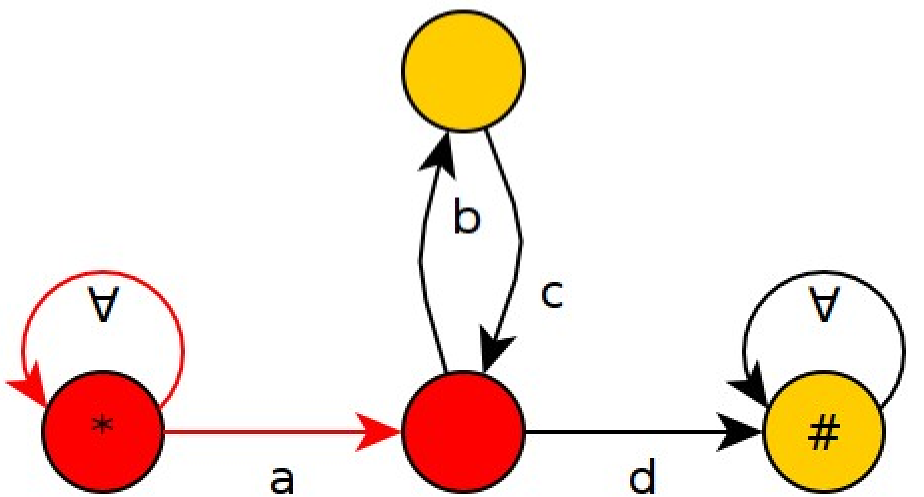
Example

xyzabcbcdxyz



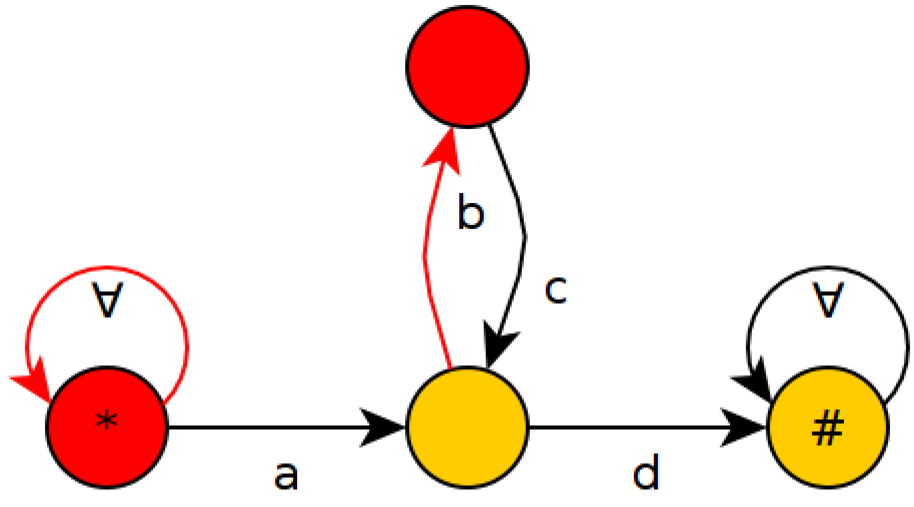
Example

xyzabcbcdxyz



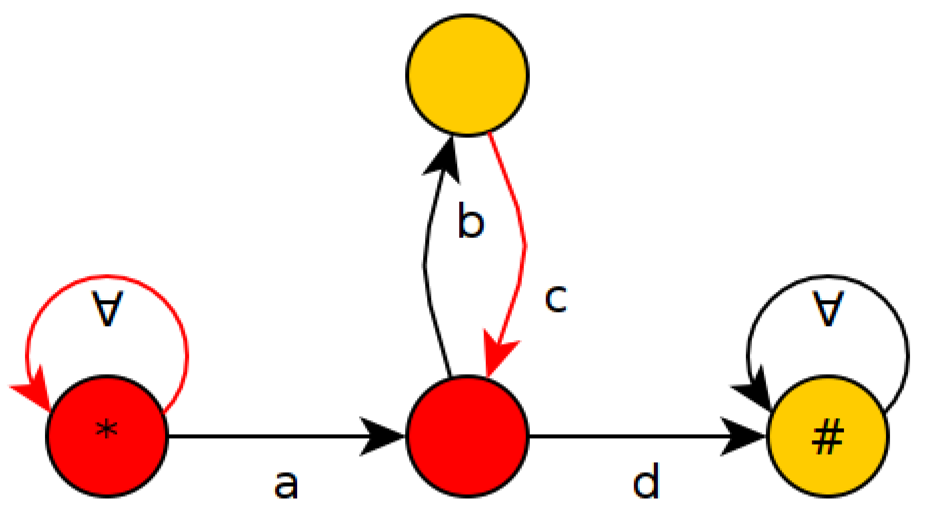
Example

xyzabcbcdxyz



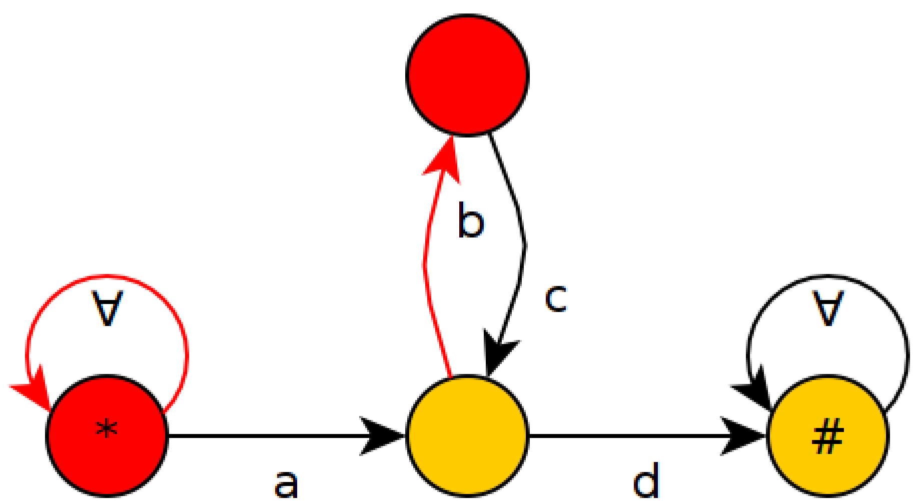
Example

xyzabcbcdxyz



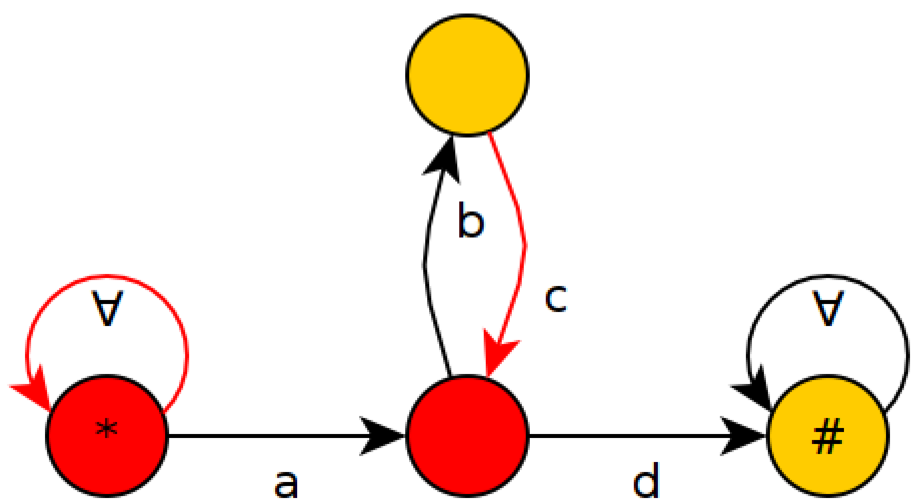
Example

xyzabcbcdxyz



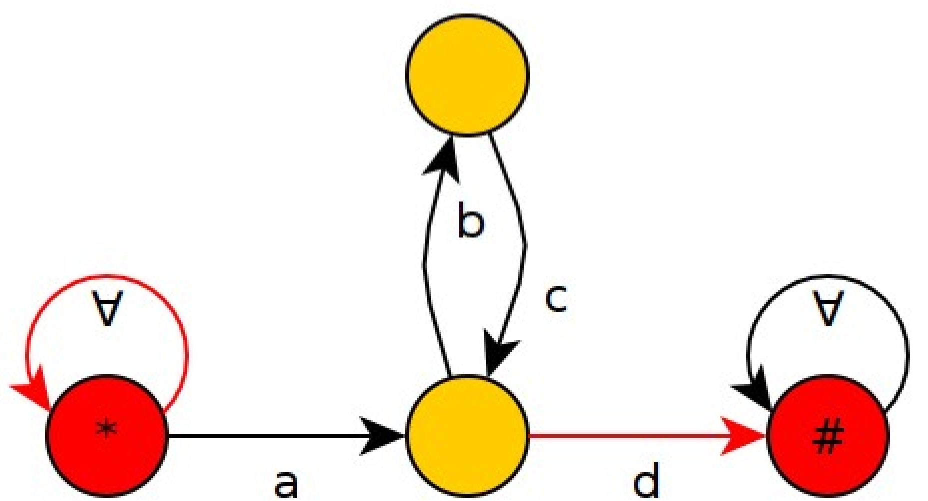
Example

xyzabcbcdxyz



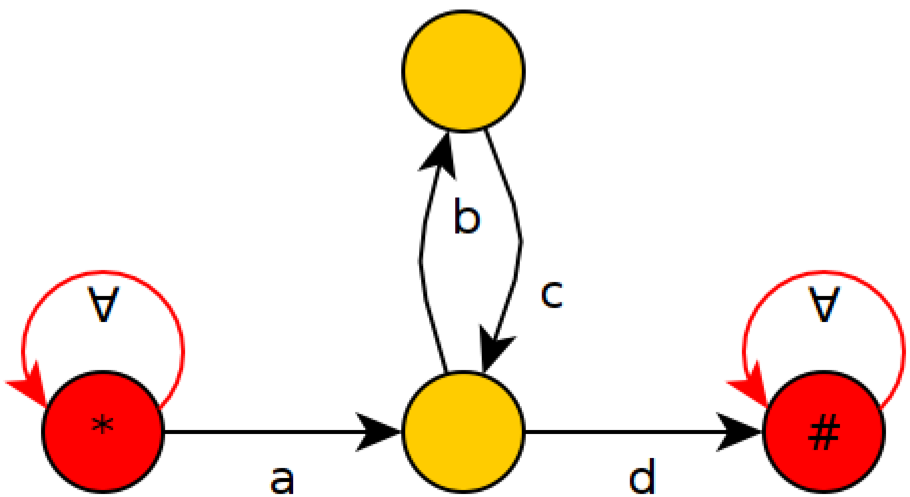
Example

xyzabcbcdxyz



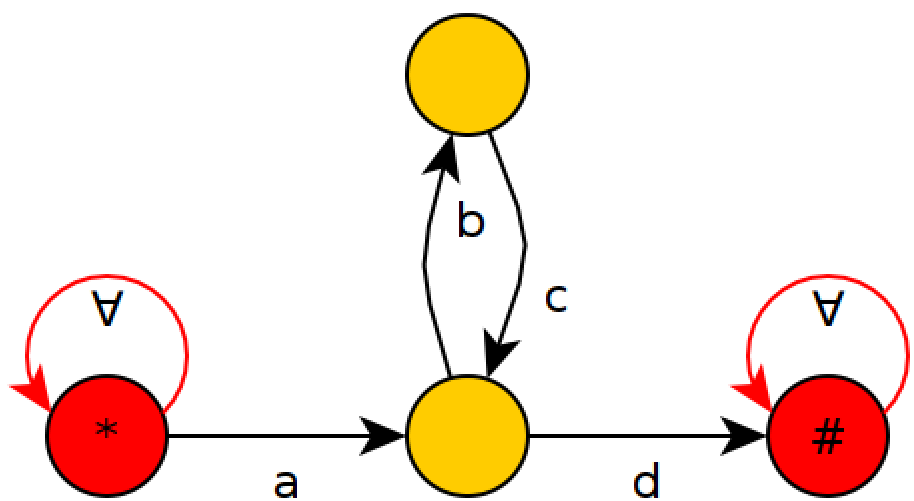
Example

xyzabcbcdxyz



Example

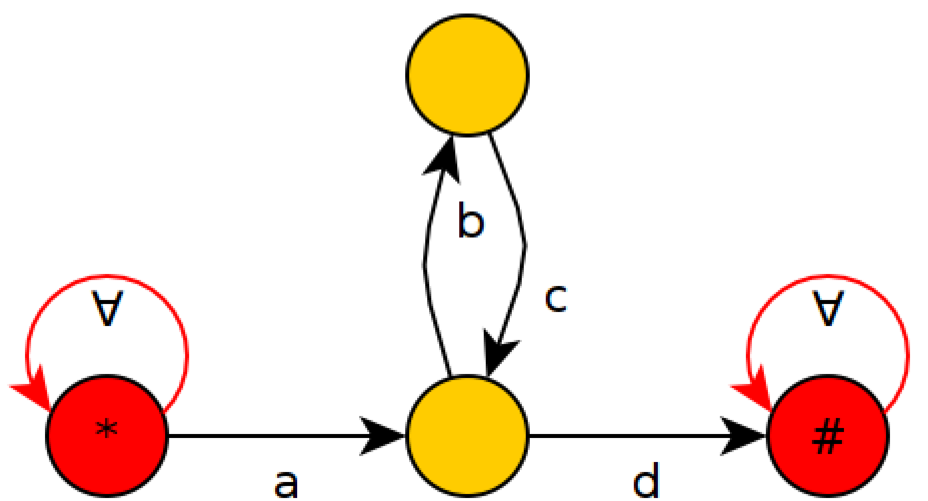
xyzabcbcdxyz



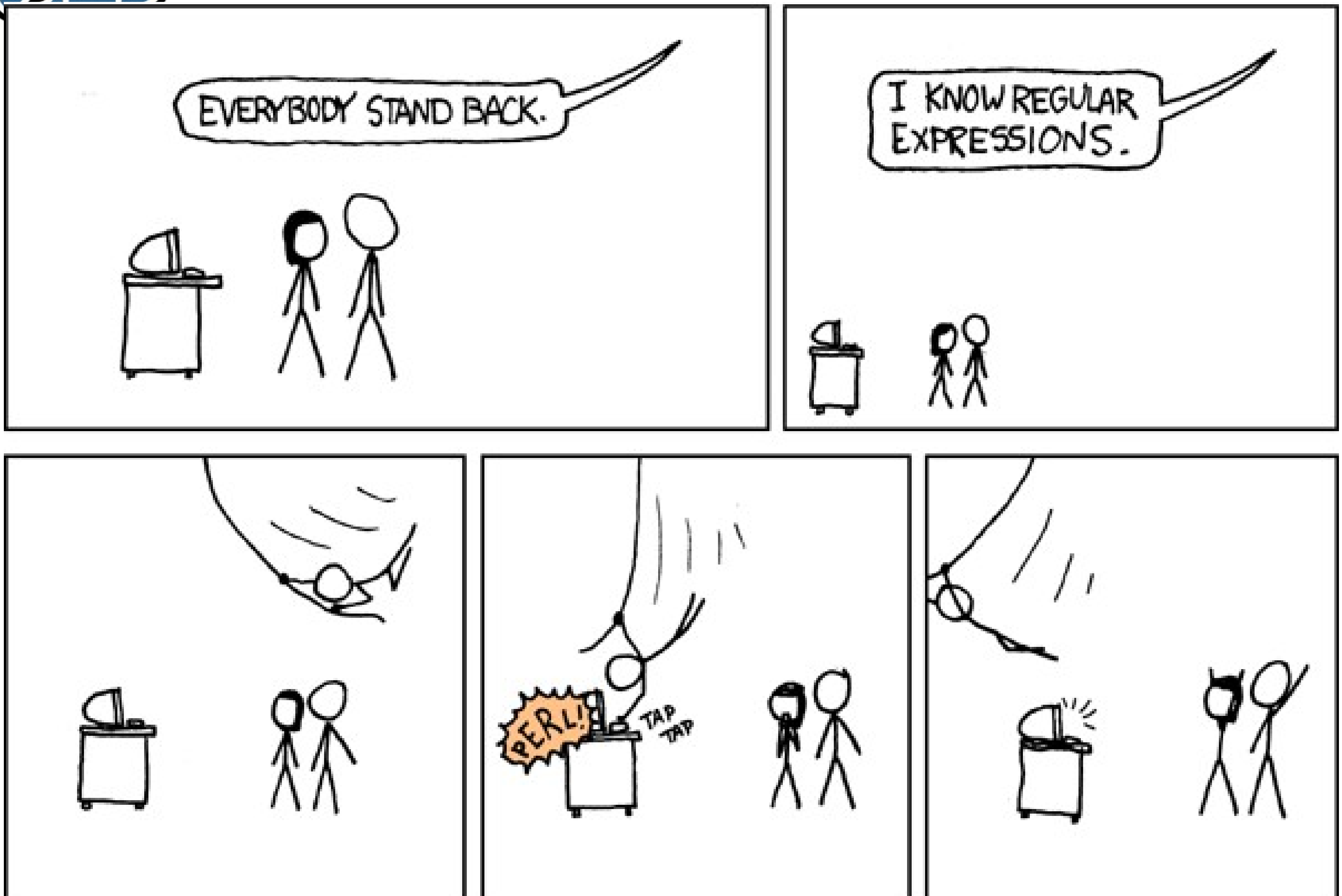
Example

xyzabcbcdxyz

Finish! Match!



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

pg_trgm

1. "regular expressions",
2. "expressive speach",
3. "regular speaker"

=>

```
'  e': {1,2}
'  r': {1,3}
'  s': {2,3}
' ex': {1,2}
' re': {1,3}
' sp': {2,3}
'ach': {2}
'ake': {3}
'ar ': {1,3}
'ch ': {2}
'eac': {2}
'eak': {3}
'egu': {1,3}
'er ': {3}
'ess': {1,2}
'exp': {1,2}
'gul': {1,3}
'ion': {1}
'ive': {2}
'ker': {3}
'lar': {1,3}
'ns ': {1}
'ons': {1}
'pea': {2,3}
'pre': {1,2}
'reg': {1,3}
'res': {1,2}
'sio': {1}
'siv': {2}
'spe': {2,3}
'ssi': {1,2}
'ula': {1,3}
've ': {2}
'xpr': {1,2}
```

Q-grams frequencies

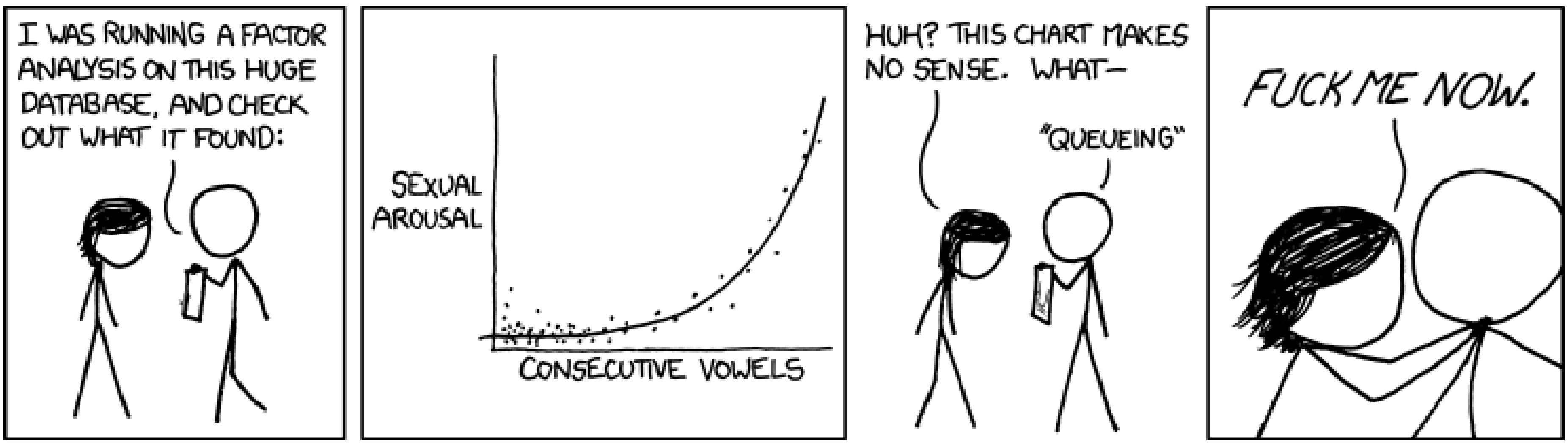
From 2.5M of DBLP paper titles

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

"Zzzzzzzzzzzzzzzzzzzzzzzzzz" 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



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}

	acd	bcd	cde	(acd OR bcd) AND cde	recheck
1	t	f	f	f	
2	f	t	t	t	f
3	f	t	f	f	
4	t	t	t	t	t
5	t	f	f	f	
6	f	f	t	f	

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

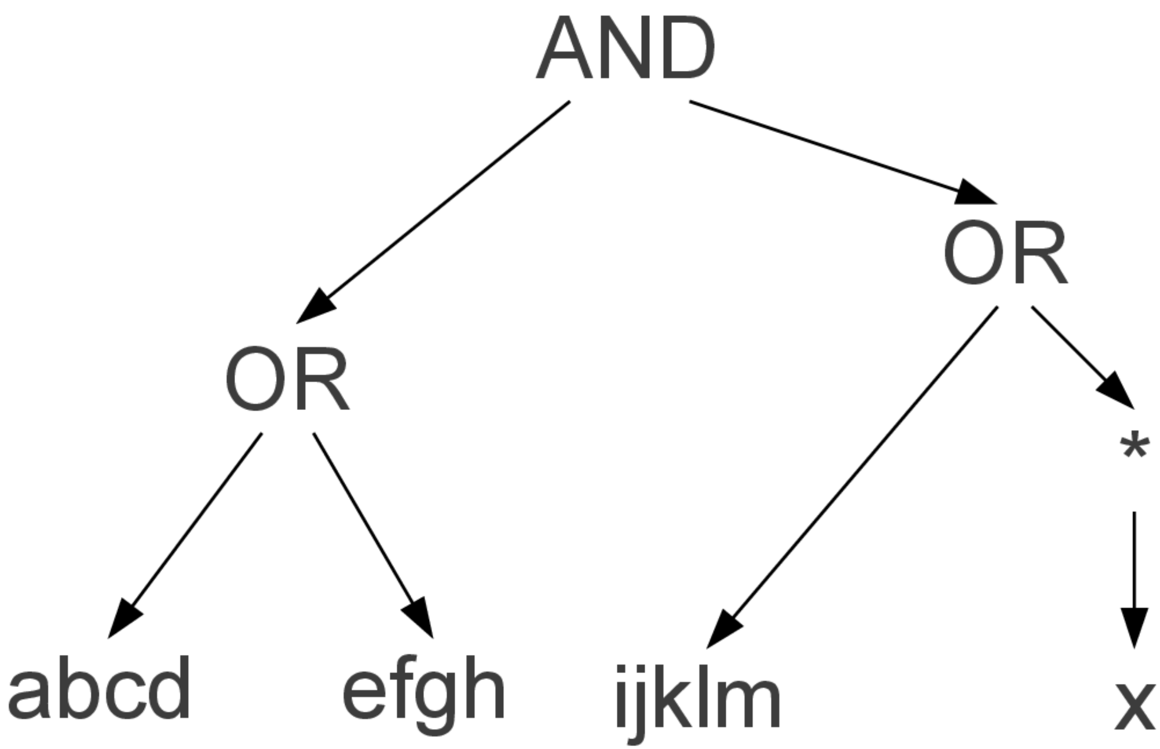
Still widely referenced as state of art work about indexing for regular expressions.

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

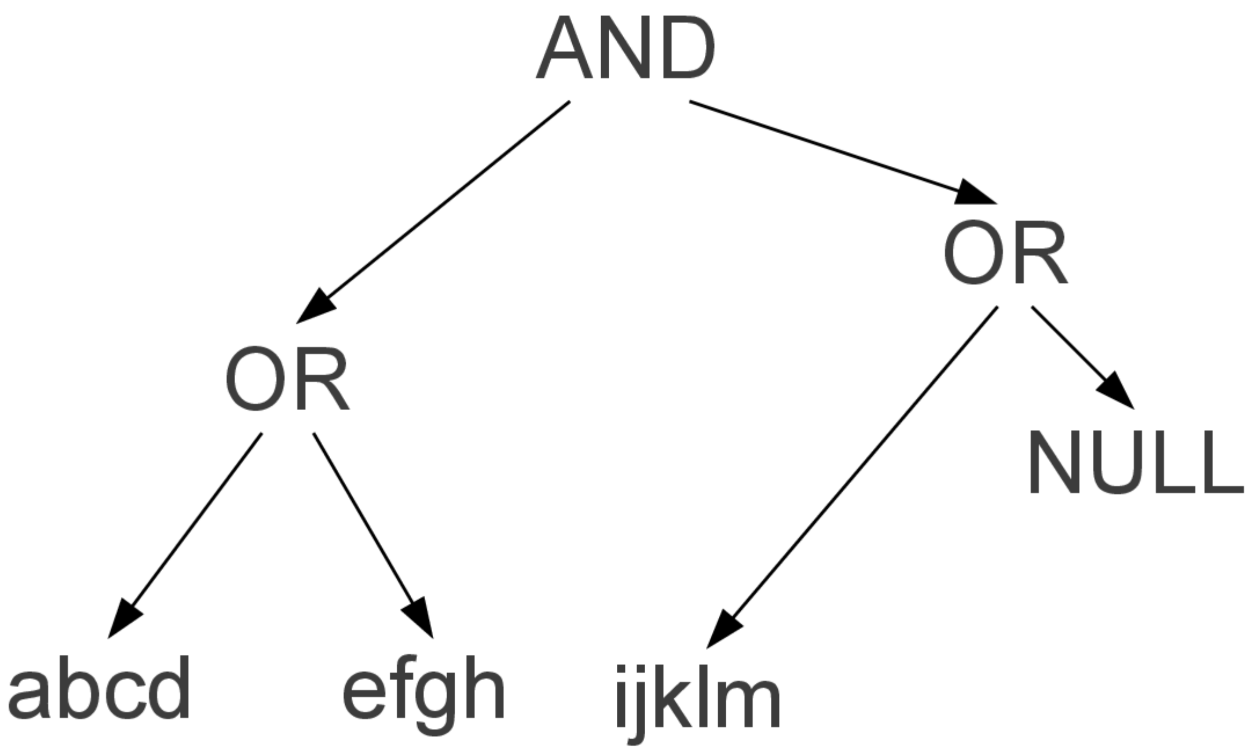
FREE method: example

Tree for /(abcd|efgh)(ijklm|x*)/



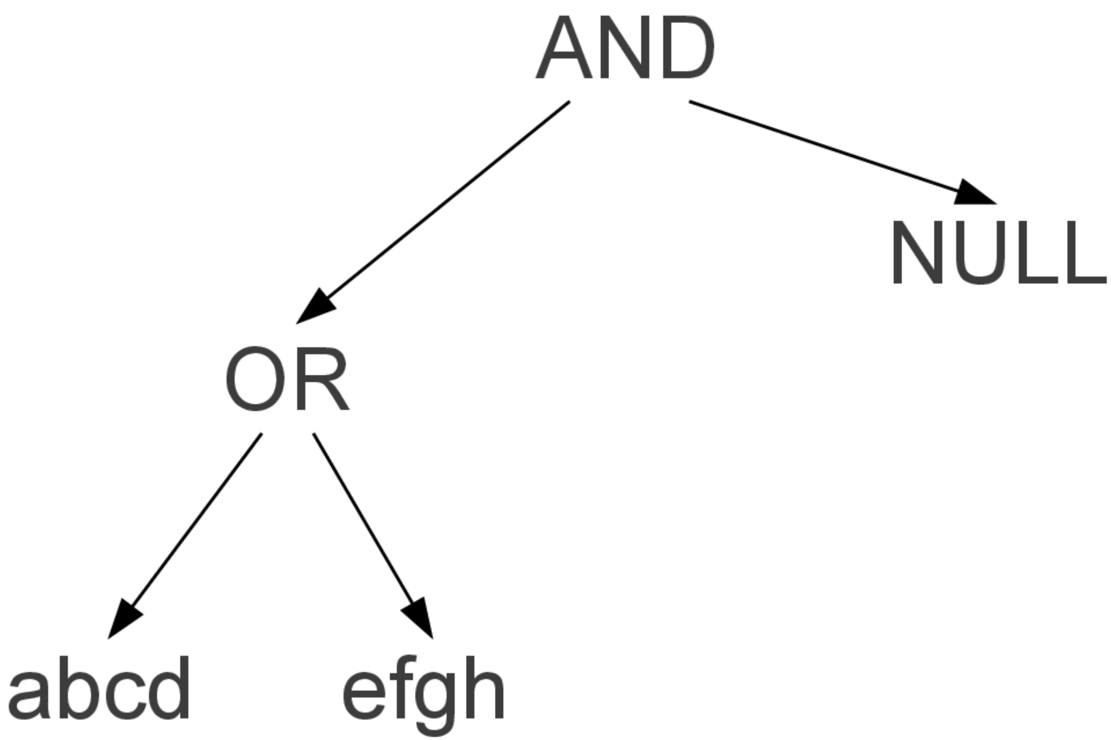
Scholar paper: example

Replace "*" nodes with NULL



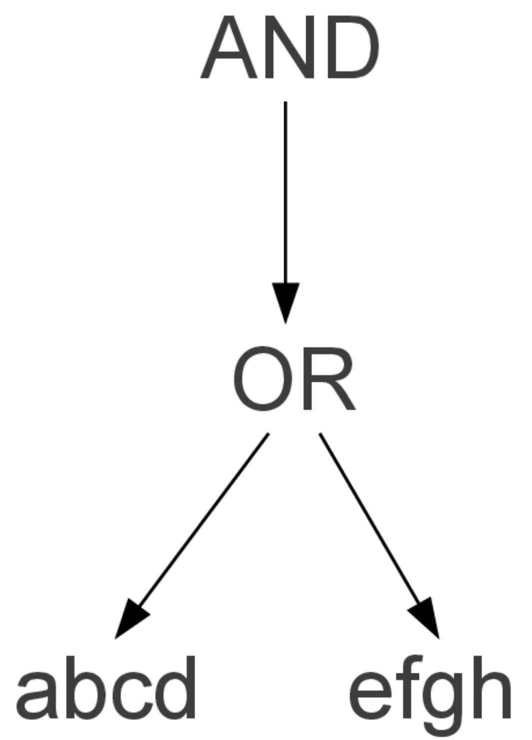
Scholar paper: example

NULL “eats” parent OR node



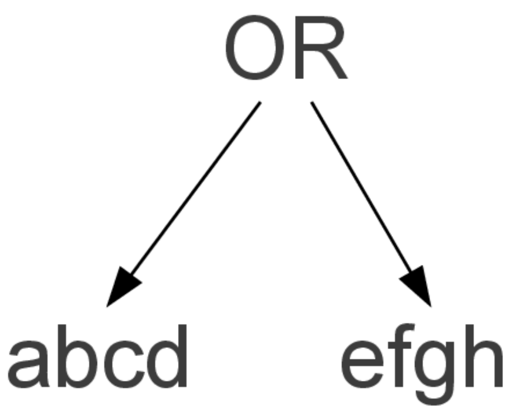
Scholar paper: example

AND node “eats” child NULL



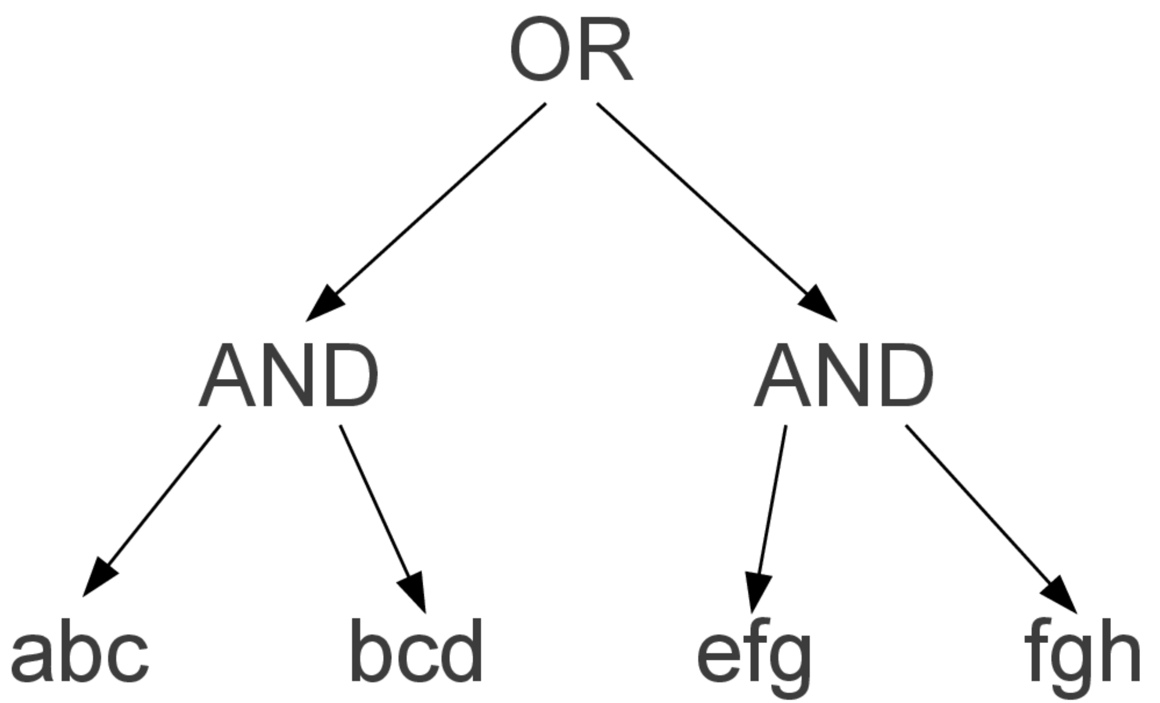
Scholar paper: example

Simplify a bit



Scholar paper: example

Expand continuous string fractions into trigrams



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

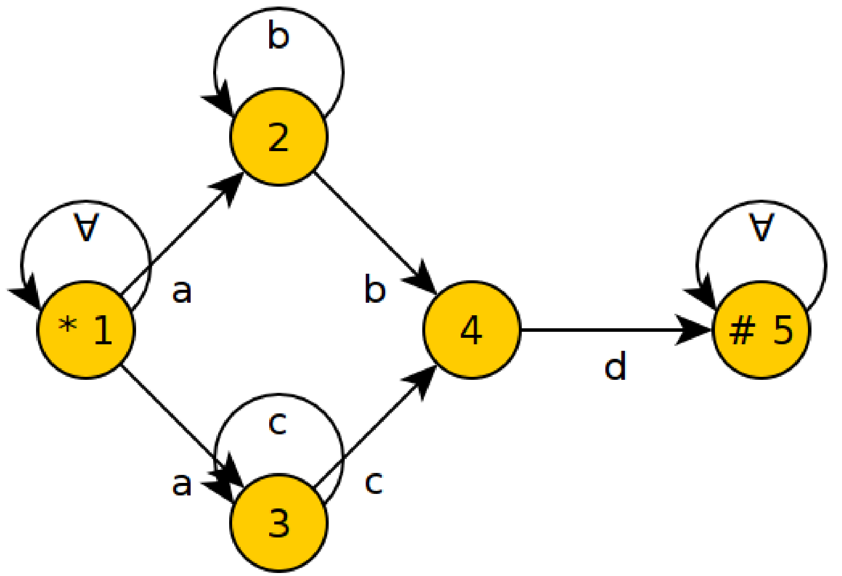
Proposed method

Proposed method

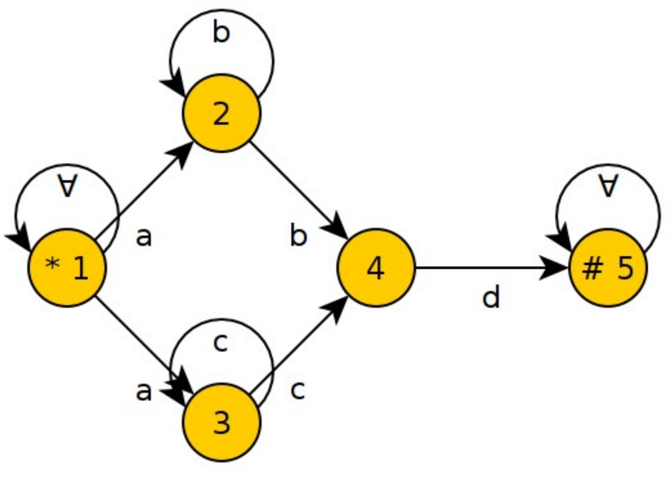
- Transform automaton into automaton like graph on trigrams
- Simplify that graph
- Use pg_trgm indexes

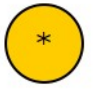
Transformation example

/a(b+|c+)d/

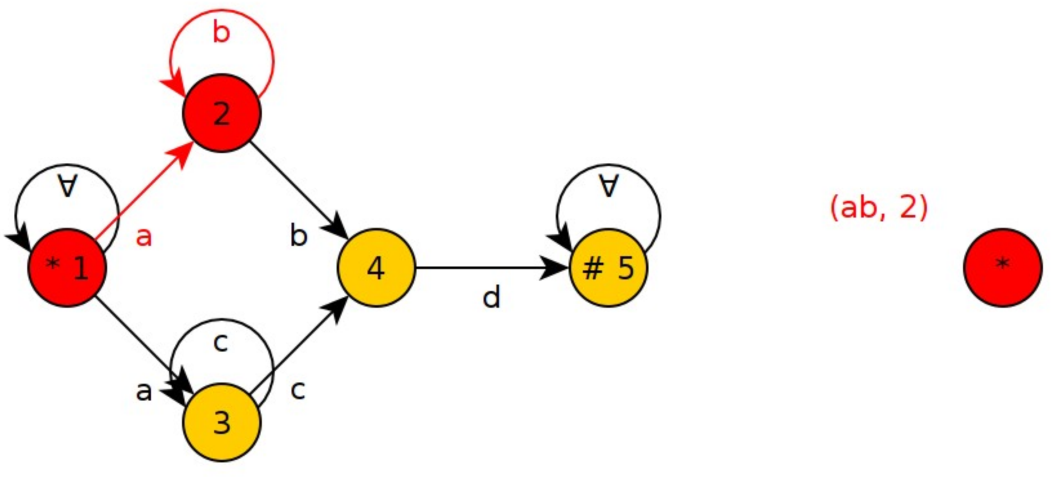


Transformation example

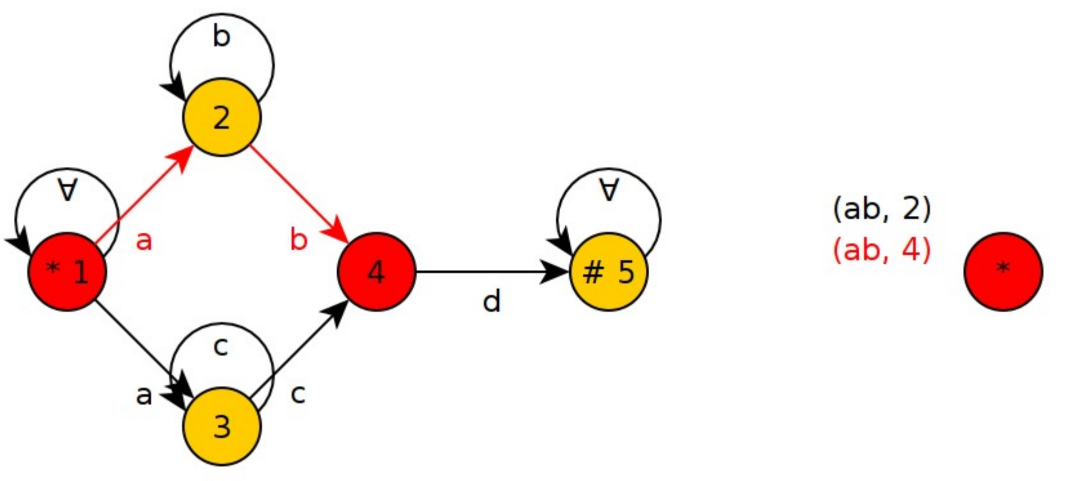




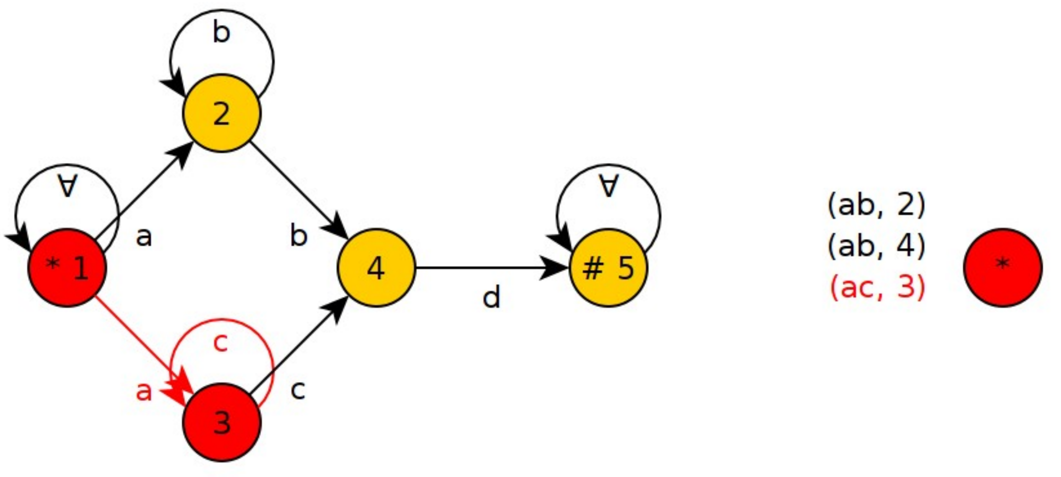
Transformation example



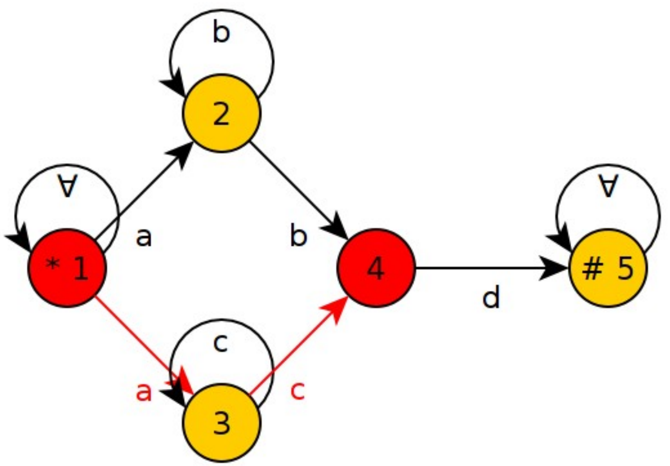
Transformation example



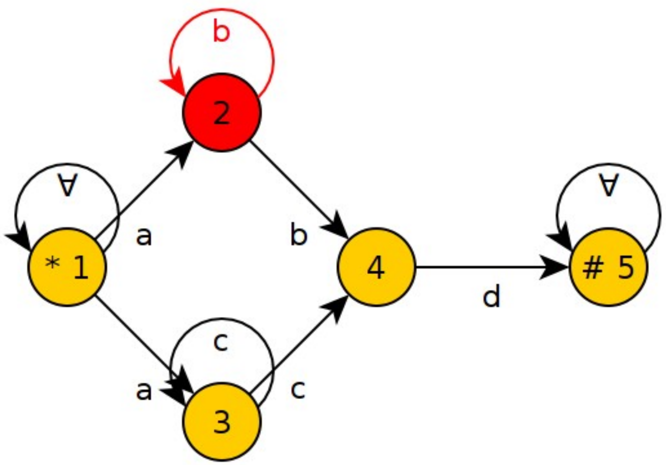
Transformation example

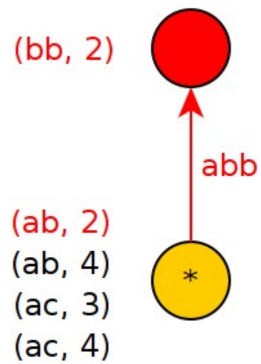


Transformation example

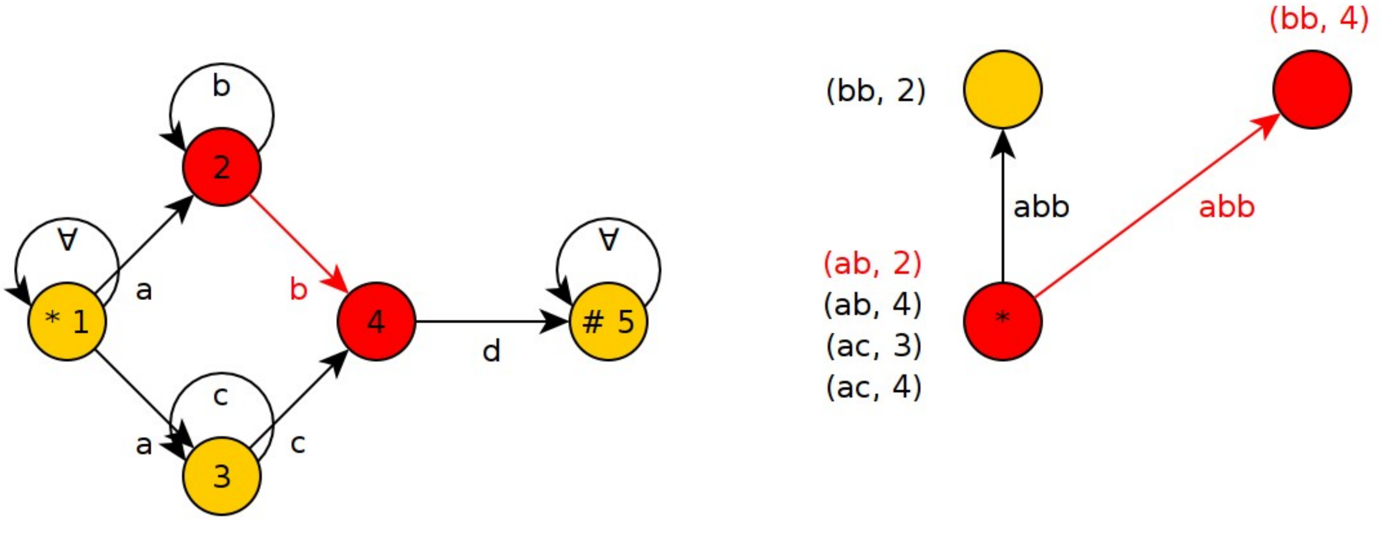


Transformation example

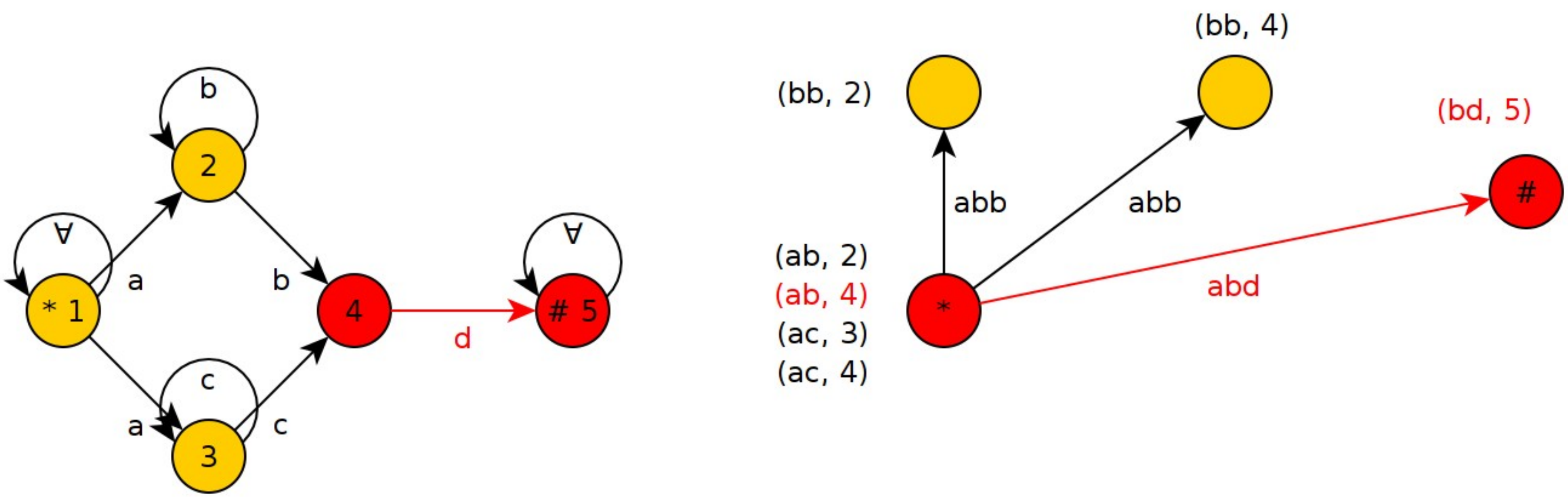




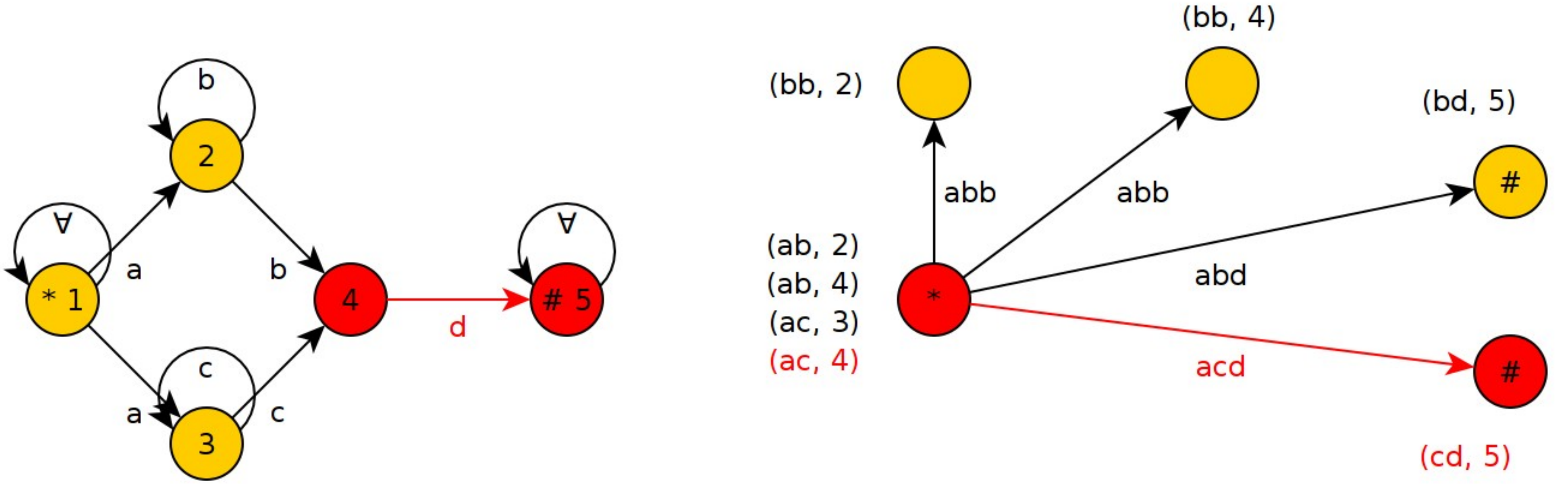
Transformation example



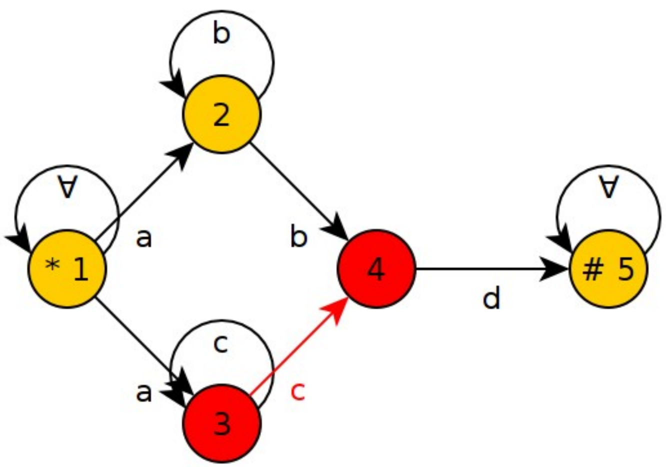
Transformation example

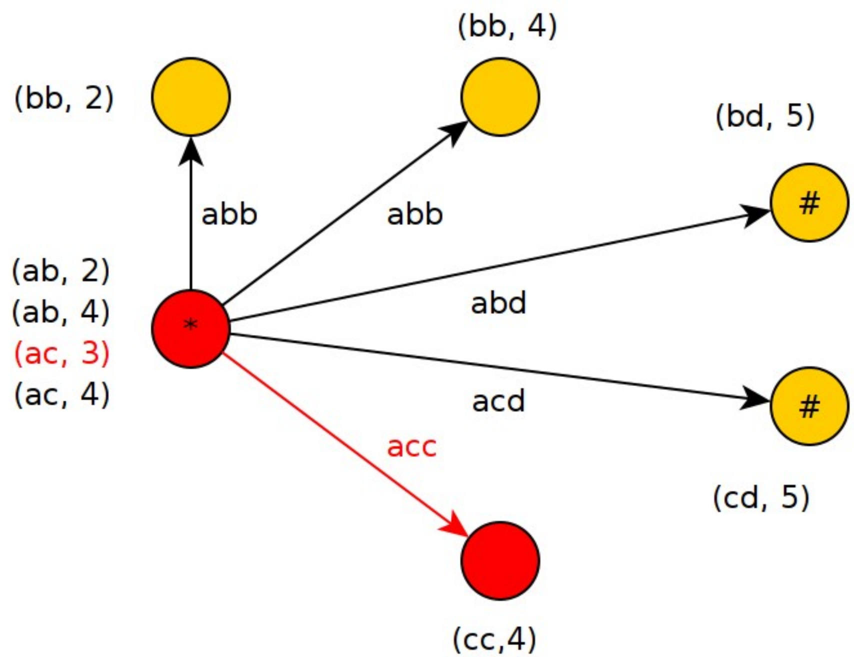


Transformation example

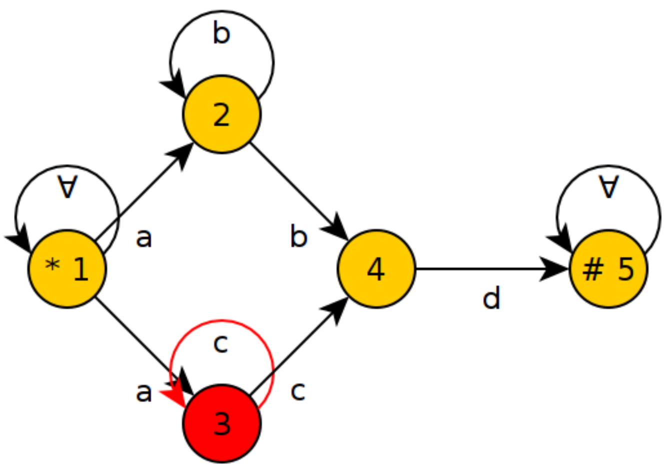


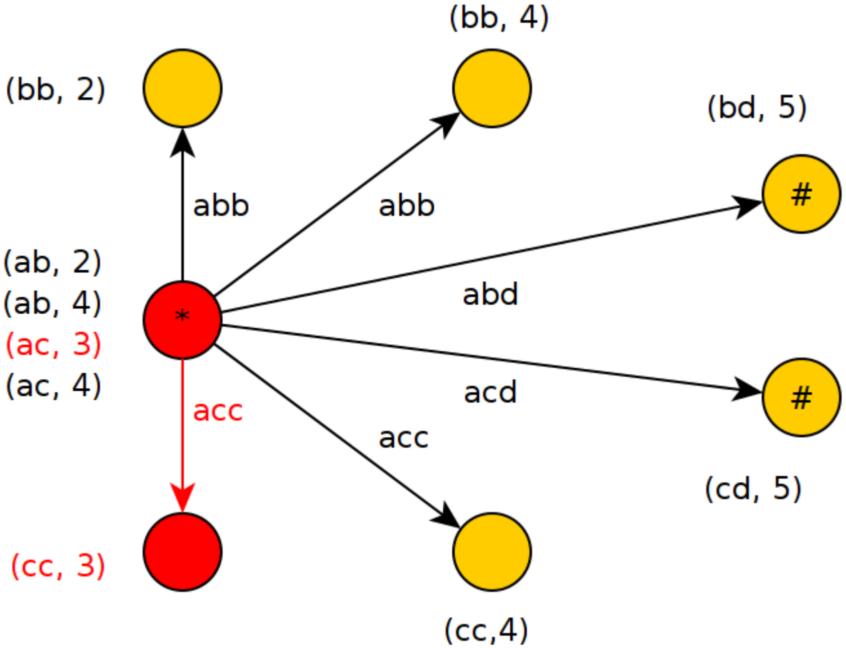
Transformation example



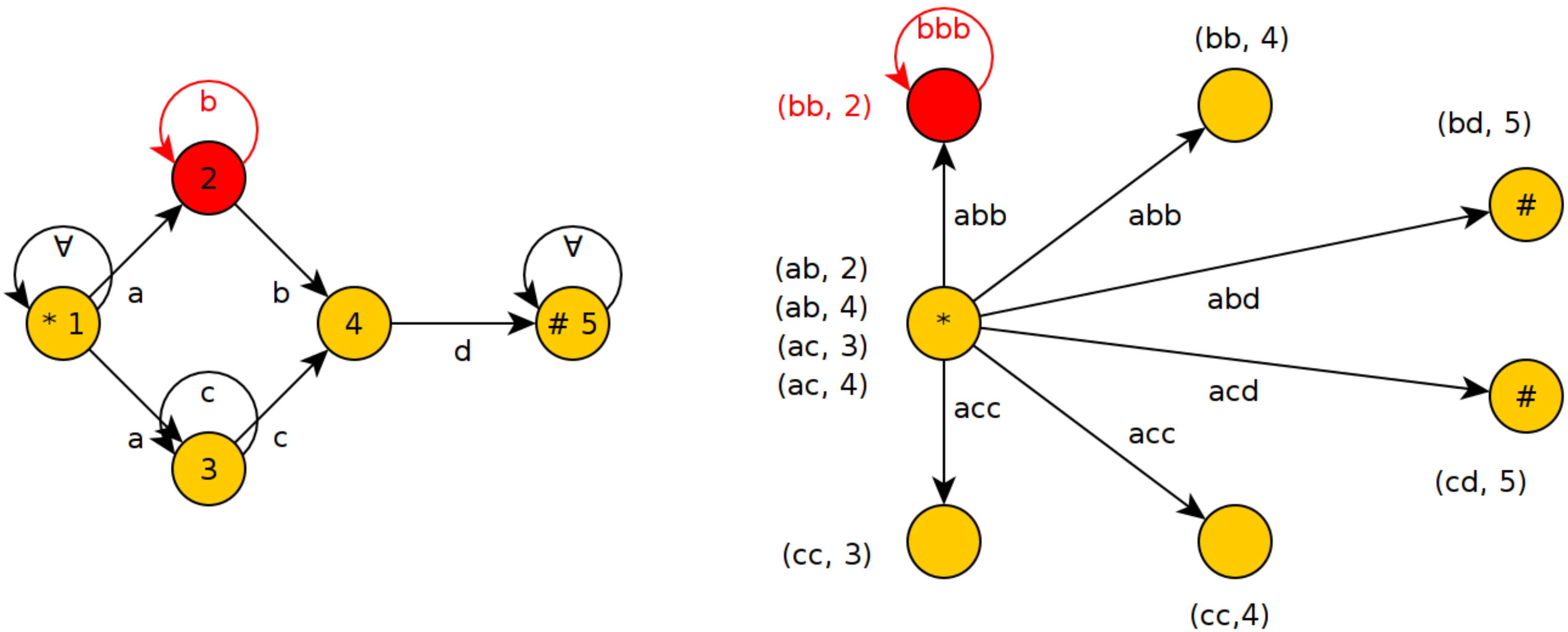


Transformation example

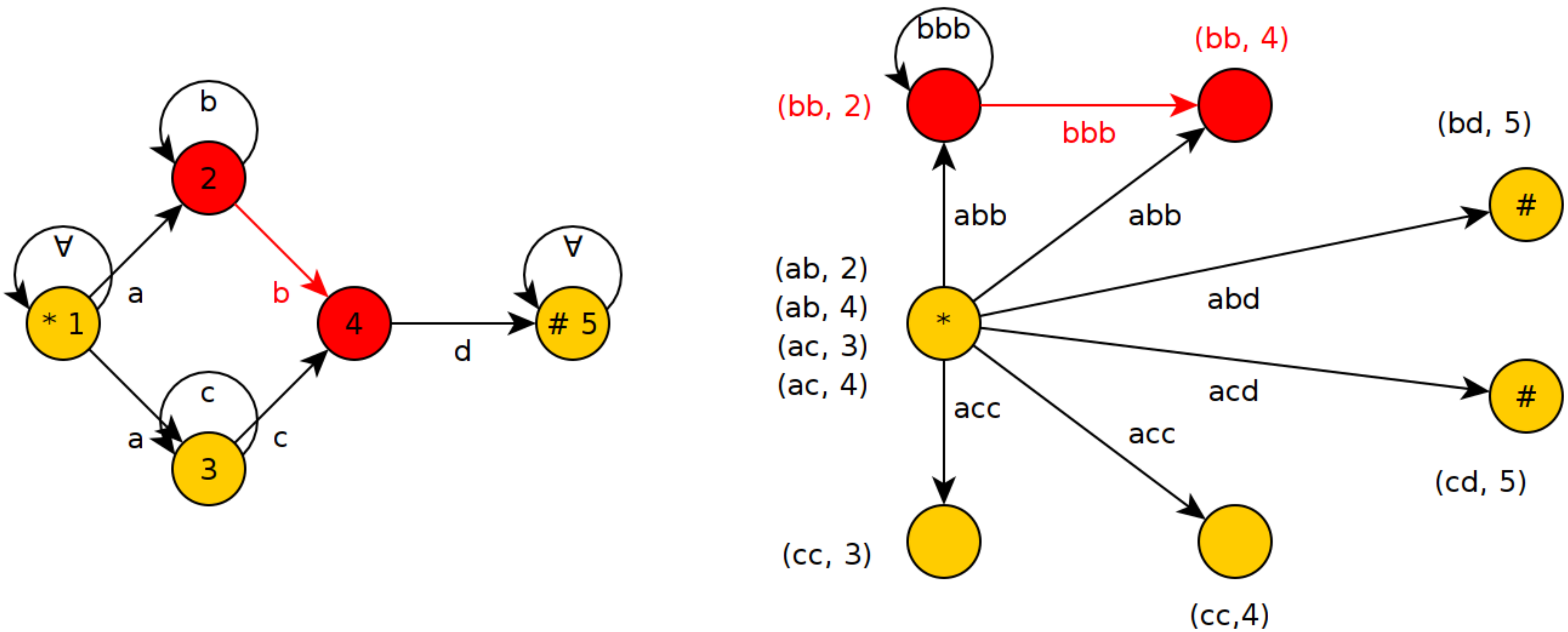




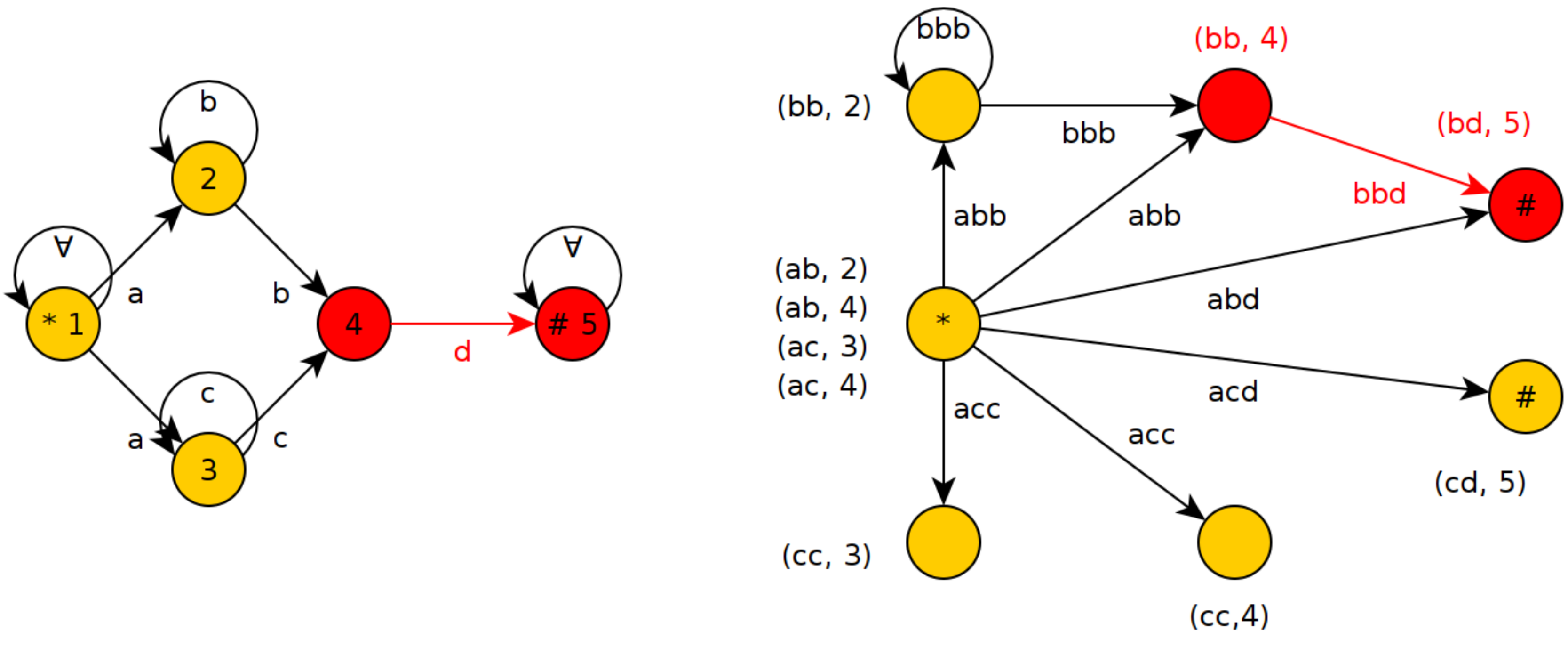
Transformation example



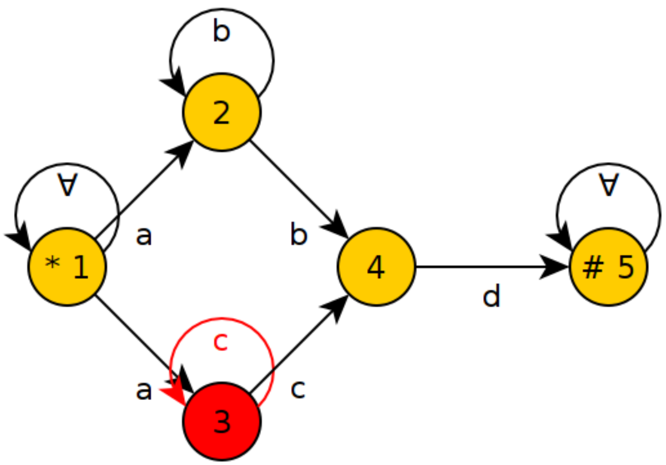
Transformation example

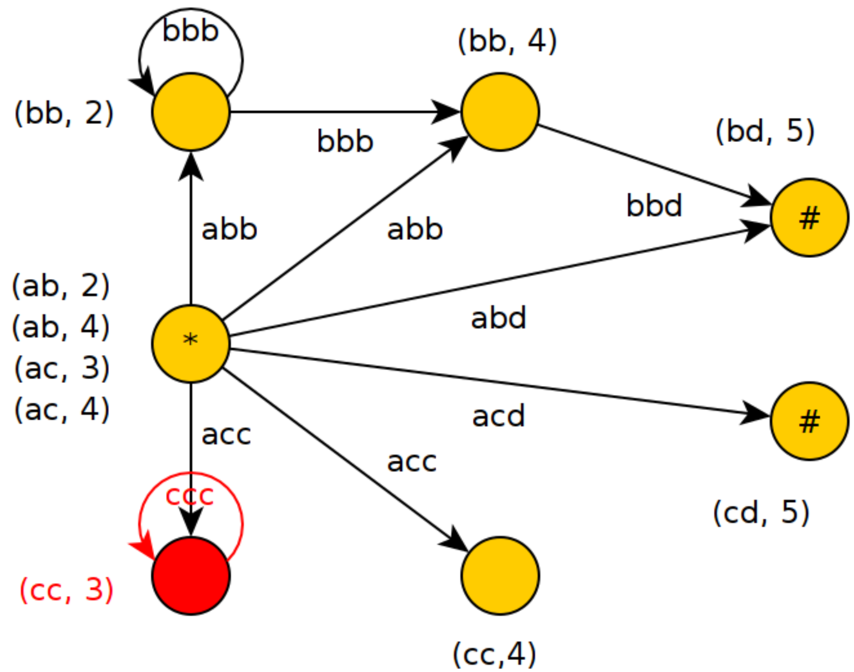


Transformation example

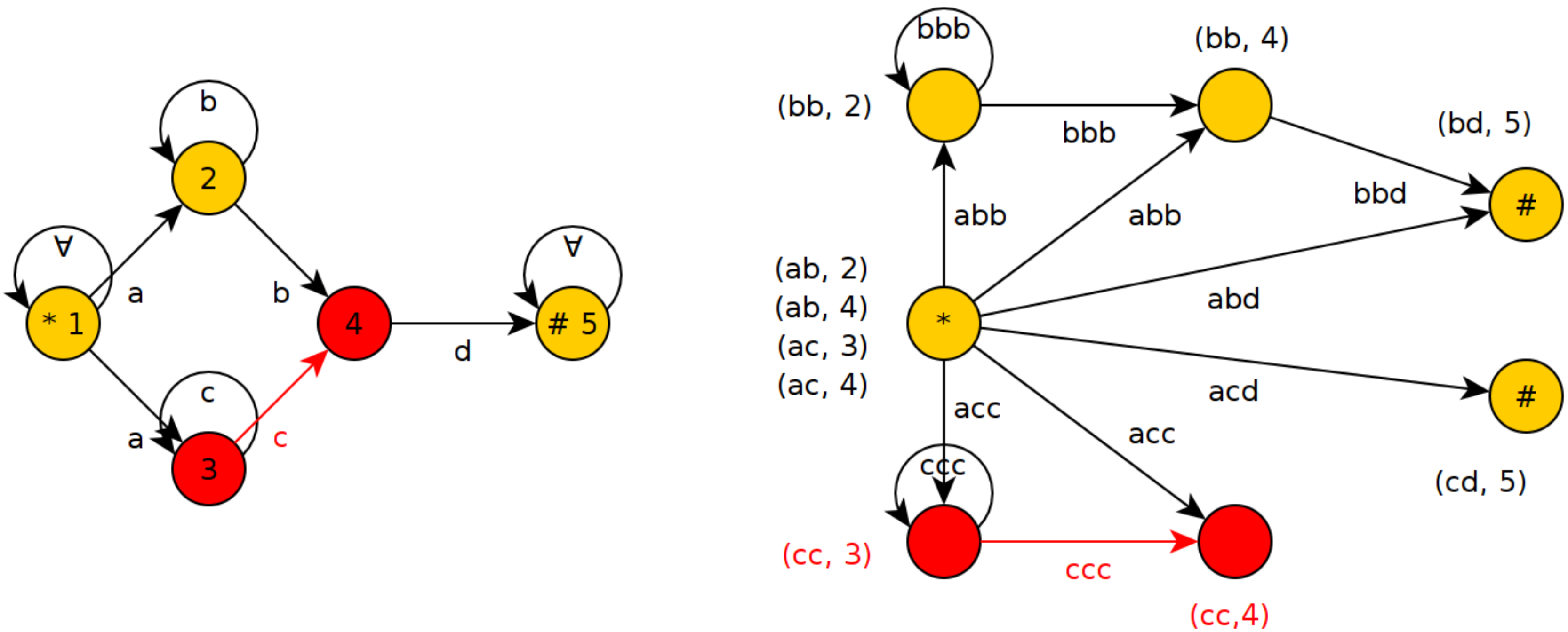


Transformation example

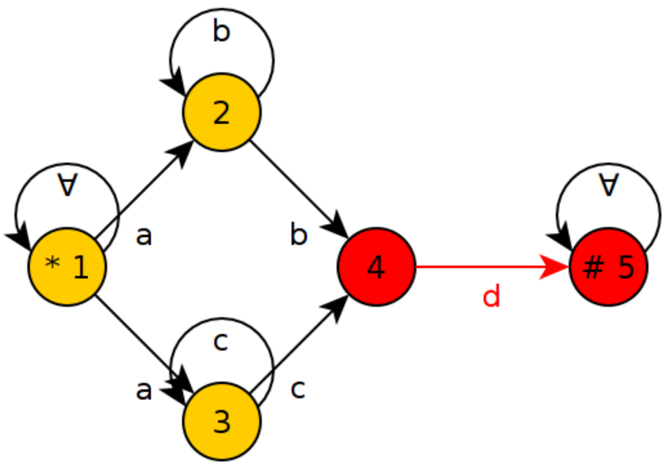


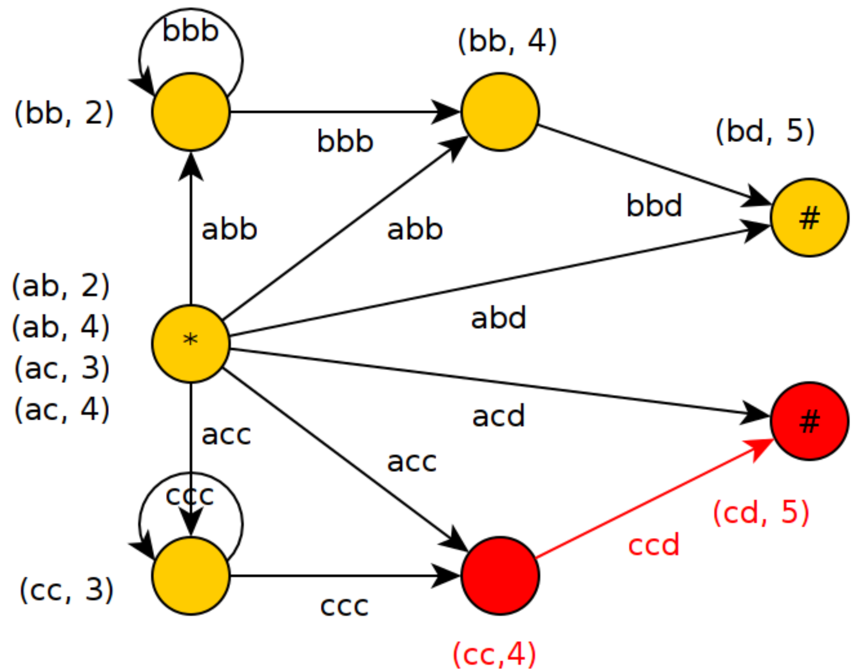


Transformation example



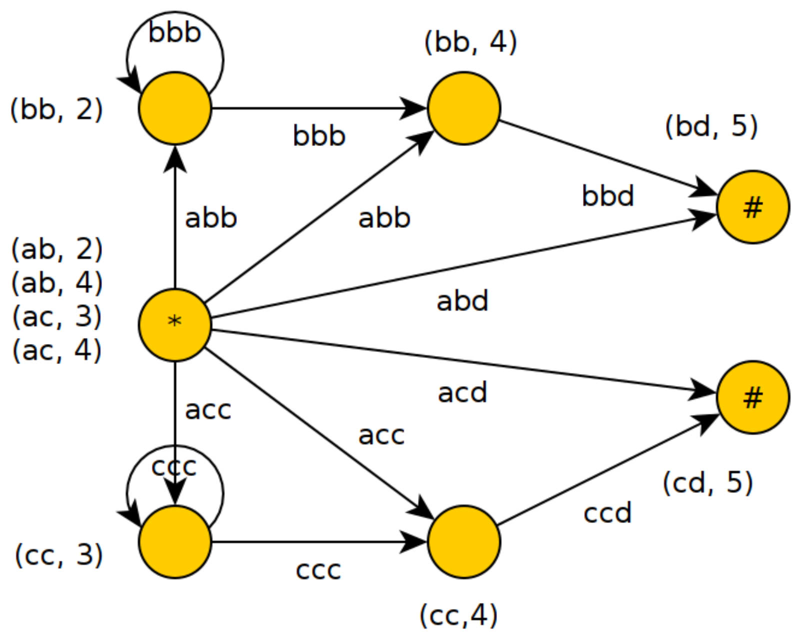
Transformation example

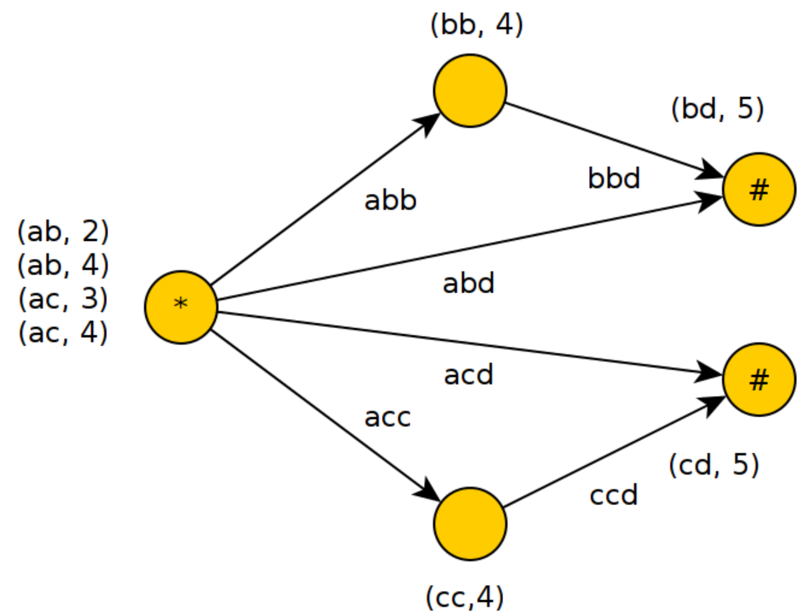




Transformation example

Result could be simplified





Transformation example

Implemented simplification technique:

collect path matrix.

abd	abb	bbd	acd	acc	ccd
1	0	0	0	0	0
0	1	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	1

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

Regex	Index scan	Seq scan
/database.*(sql|query)/	773 ms	18653 ms
/postgres(ql)?/	268 ms	17574 ms
/plan+er/	253 ms	12885 ms
/(nucl|anino).*acid/	200 ms	20085 ms
/[aei](bc)+a/	2 ms	13195 ms

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 simplier)
- Encoding problem is solved.

New patch will be posted soon.

GIN improvements

Summary of changes

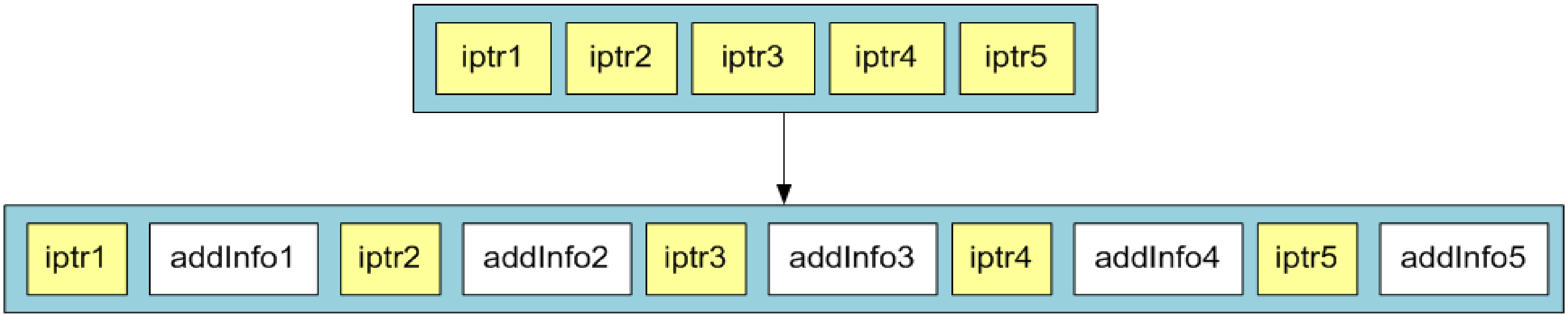
- Compressed storage with additional information
- Optimized search («frequent_entry & rare_entry» case)
- Return ordered results by index (ORDER BY optimization)

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 infromation

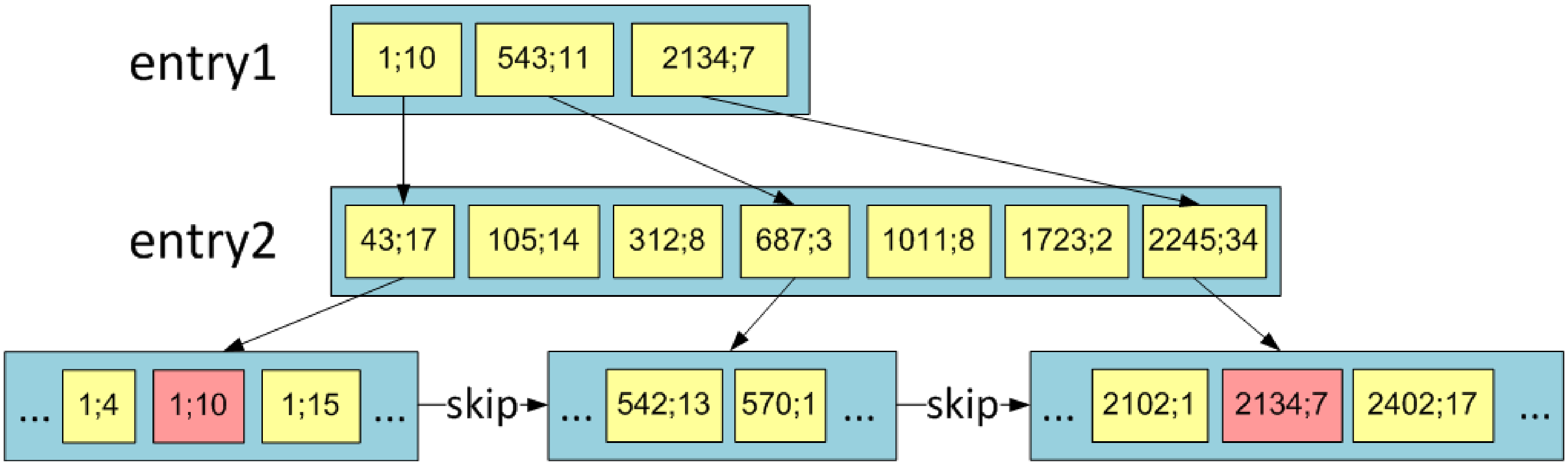


Use increments and variable byte encoding to keep index small

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

Fast scan

entry1 && entry2

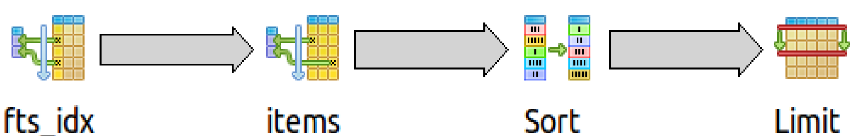


Visiting 3 pages instead of 7

ORDER BY using index

Before

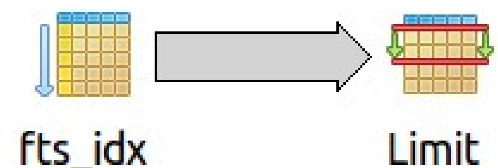
SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY
ts_rank(fts, to_tsquery('english', 'query')) DESC
LIMIT 10;



Ranking and sorting are outside the fulltext index

After

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



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

368 ms vs 13 ms

Example: frequent entry (30%)

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	367.687 ms	94.6 %
Bitmap Index Scan	1	6.570 ms	1.7 %
Limit	1	0.001 ms	0.0 %
Sort	1	14.465 ms	3.7 %

388 ms

After:

node type	count	sum of times	% of query
Index Scan	1	13.346 ms	100.0 %
Limit	1	0.001 ms	0.0 %

13 ms

Example: rare entry (0.08%)

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	0.959 ms	93.4 %
Bitmap Index Scan	1	0.027 ms	2.6 %
Limit	1	0.001 ms	0.1 %
Sort	1	0.040 ms	3.9 %

1.1 ms

After:

node type	count	sum of times	% of query
Index Scan	1	0.052 ms	98.1 %
Limit	1	0.001 ms	1.9 %

0.07 ms

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

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	1.547 ms	23.0 %
Bitmap Index Scan	1	5.151 ms	76.7 %
Limit	1	0.000 ms	0.0 %
Sort	1	0.022 ms	0.3 %

6.7 ms

After:

node type	count	sum of times	% of query
Index Scan	1	0.998 ms	100.0 %
Limit	1	0.000 ms	0.0 %

1.0 ms

Thank you for attention!
Sponsors are welcome!