Pattern Matching and Full Text Search in SQL

Markus Schatten, PhD

Assistant Professor
Faculty of Organization and Informatics,
University of Zagreb
Pavlinska 2, 42000 Varaždin
markus.schatten@foi.hr

17.02.2012.

Outline

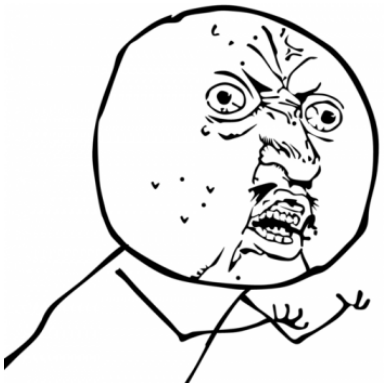
Tools

- PostgreSQL 9.1

Tools

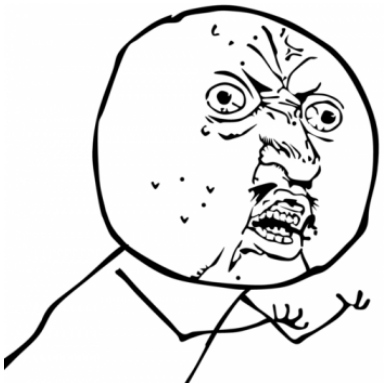
- PostgreSQL 9.1
- LibreOffice.org Base
  - Connect with `host=localhost dbname=wikileaks port=5432`
    username `ubuntu`
    password `wikileaks`

Why pattern matching in databases?



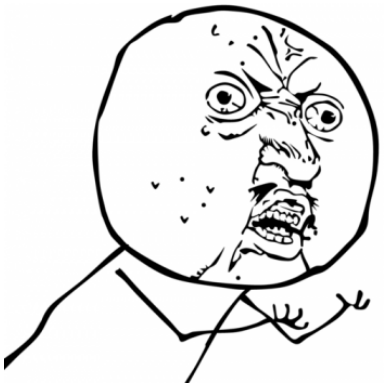
- “Good” databases often do not need pattern matching

Why pattern matching in databases?



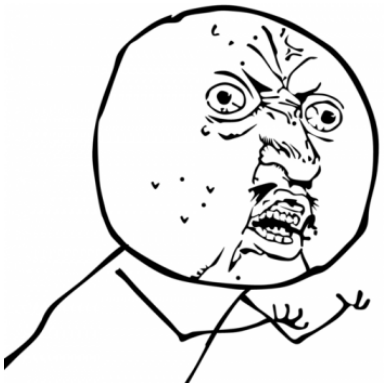
- “Good” databases often do not need pattern matching
- A curious case:
  - field name “`data`”
  - type `varchar( 200 )`
  - structure: first 10 characters internal user ID (numeric), then user’s surname ended by a ’$’ sign, then user’s name again ended by a ’$’ sign, one character indicator of user type ...

Why pattern matching in databases?



- "Good" databases often do not need pattern matching
- A curious case:
  - field name "`data`"
  - type `varchar( 200 )`
  - structure: first 10 characters internal user ID (numeric), then user's surname ended by a '$' sign, then user's name again ended by a '$' sign, one character indicator of user type ...
- Databases with lots of textual information (web applications, document management systems, ...)

Why pattern matching in databases?



- "Good" databases often do not need pattern matching
- A curious case:
  - field name "`data`"
  - type `varchar( 200 )`
  - structure: first 10 characters internal user ID (numeric), then user's surname ended by a '$' sign, then user's name again ended by a '$' sign, one character indicator of user type ...
- Databases with lots of textual information (web applications, document management systems, ...)
- ...

Outline

The `LIKE` expression

- Standard SQL pattern matching expression

The `LIKE` expression

- Standard SQL pattern matching expression
- Uses simple wildcard characters to match strings
  - `%` matches any 0 or more characters
  - `_` matches exactly one character

The `LIKE` expression

- Standard SQL pattern matching expression
- Uses simple wildcard characters to match strings
  - `%` matches any 0 or more characters
  - `_` matches exactly one character
- Usefull for keyword search and (very) simple patterns

```
SELECT 'e-Discovery' LIKE '%e%very%'

SELECT 'e-Discovery' LIKE '%Disco%'

SELECT 'e-Discovery' LIKE '___is______'

SELECT 'e-Discovery' NOT LIKE '_e%'

SELECT 'e-Discovery' NOT LIKE '%asy%'

SELECT 'e-Discovery' LIKE '%over_'
```

```
SELECT *
FROM cable
WHERE origin LIKE '%ZAGREB%'

SELECT *
FROM cable
WHERE content LIKE '%TELEFONICA%'
```

The `ILIKE` expression

- PostgreSQL specific pattern matching expression (not standard!)

The `ILIKE` expression

- PostgreSQL specific pattern matching expression (not standard!)
- Same functionality as `LIKE` but case insensitive

```
SELECT *
FROM cable
WHERE origin ILIKE '%zagreb%'

SELECT *
FROM cable
WHERE content ILIKE '%telefonica%'
```

LIKE and ILIKE

- Shortcuts:
  - `~~ = LIKE`
  - `!~~ = NOT LIKE`
  - `~~* = ILIKE`
  - `!~~* = NOT ILIKE`

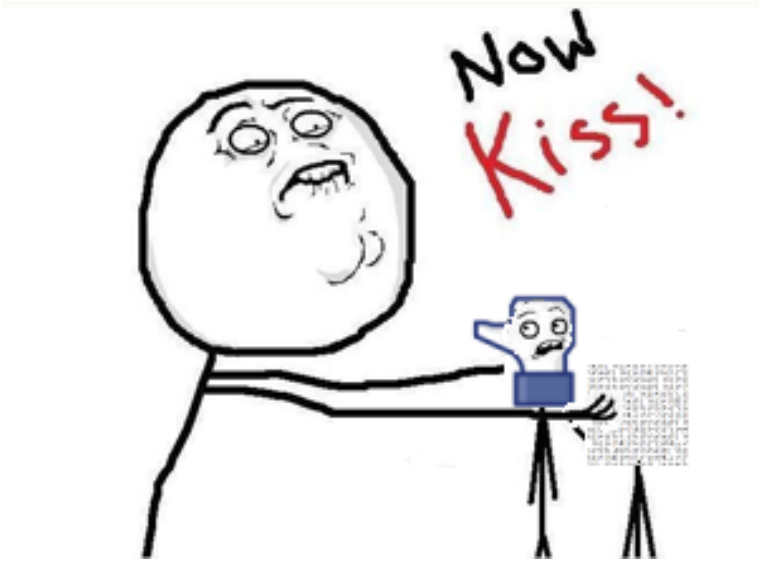
LIKE and ILIKE

- Shortcuts:
  - `~~ = LIKE`
  - `!~~ = NOT LIKE`
  - `~~* = ILIKE`
  - `!~~* = NOT ILIKE`
- Advantage - very fast

LIKE and ILIKE

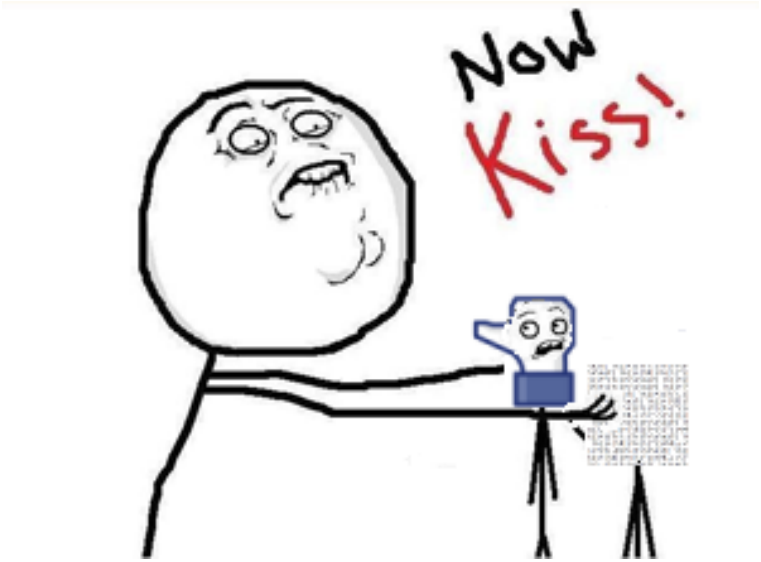
- Shortcuts:
  - `~~ = LIKE`
  - `!~~ = NOT LIKE`
  - `~~* = ILIKE`
  - `!~~* = NOT ILIKE`
- Advantage - very fast
- Dissadvantage - very limited

The `SIMILAR TO` expression



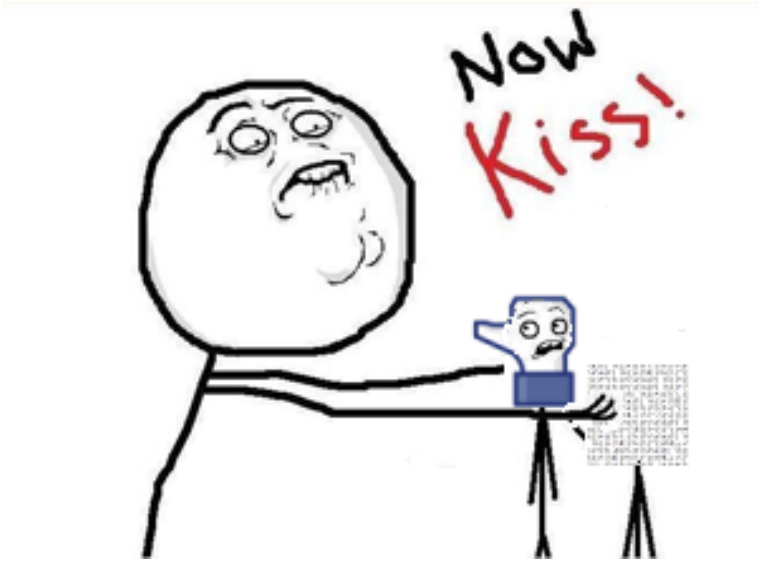
- "a curious cross between `LIKE` notation and common regular expression notation"

The `SIMILAR TO` expression



- "a curious cross between `LIKE` notation and common regular expression notation"
- Uses SQL standard definition of regular expressions

The `SIMILAR TO` expression



- "a curious cross between `LIKE` notation and common regular expression notation"
- Uses SQL standard definition of regular expressions
- Like the `LIKE` expression always matches the whole string

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters
- `_` matches exactly one character

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters
- `_` matches exactly one character
- `|` denotes alternation (either of two alternatives)

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters
- `_` matches exactly one character
- `|` denotes alternation (either of two alternatives)
- `*` denotes repetition of the previous item zero or more times

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters
- `_` matches exactly one character
- `|` denotes alternation (either of two alternatives)
- `*` denotes repetition of the previous item zero or more times
- `+` denotes repetition of the previous item one or more times

SIMILAR TO metacharacters

- % matches any 0 or more characters
- _ matches exactly one character
- | denotes alternation (either of two alternatives)
- * denotes repetition of the previous item zero or more times
- + denotes repetition of the previous item one or more times
- ( ) parentheses can be used to group items into a single logical item

`SIMILAR TO` metacharacters

- `%` matches any 0 or more characters
- `_` matches exactly one character
- `|` denotes alternation (either of two alternatives)
- `*` denotes repetition of the previous item zero or more times
- `+` denotes repetition of the previous item one or more times
- `()` parentheses can be used to group items into a single logical item
- `[...]` a bracket expression specifies a character class

```
SELECT 'never' SIMILAR TO 'never'

SELECT 'gonna' NOT SIMILAR TO 'gon'

SELECT 'give' SIMILAR TO '%(i|w)%'

SELECT 'you up' NOT SIMILAR TO '(you_)|(up)'
```

```
SELECT 'email: mschatte@foi.hr'
  SIMILAR TO 'email: *%@foi.hr'

SELECT 'email:   mschatte@foi.hr'
  SIMILAR TO 'email: *%@foi.hr'
```

```
SELECT 'foooooo' SIMILAR TO 'f(o)*'

SELECT 'foooooo' SIMILAR TO 'f(o)+'

SELECT 'f' SIMILAR TO 'f(o)*'

SELECT 'f' NOT SIMILAR TO 'f(o)+'
```

Bracket expressions

`[0-9]` matches any digit (0-9)

Bracket expressions

`[0-9]` matches any digit (0-9)

`[a-z]` matches any lowercase character

Bracket expressions

- `[0-9]` matches any digit (0-9)
- `[a-z]` matches any lowercase character
- `[A-Z]` matches any uppercase character

Bracket expressions

- `[0-9]` matches any digit (0-9)
- `[a-z]` matches any lowercase character
- `[A-Z]` matches any uppercase character
- `[.?#]` matches either the '.', the '?' or the '#' character

Bracket expressions

- `[0-9]` matches any digit (0-9)
- `[a-z]` matches any lowercase character
- `[A-Z]` matches any uppercase character
- `[.?#]` matches either the '.', the '?' or the '#' character
- `[a-z3-6.,]` matches a lowercase character, a digit from 3 to 6, the '.' or the ',' character

Bracket expressions

`[0-9]` matches any digit (0-9)

`[a-z]` matches any lowercase character

`[A-Z]` matches any uppercase character

`[.?#]` matches either the '.', the '?' or the '#' character

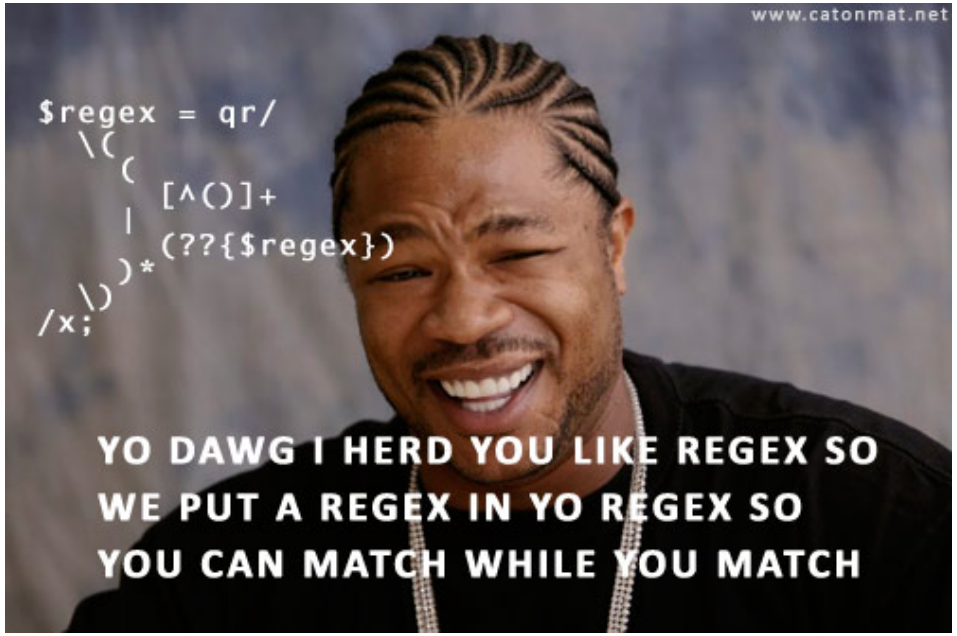
`[a-z3-6.,]` matches a lowercase character, a digit from 3 to 6, the '.' or the ',' character

...

```
SELECT '27 October 2001'
  SIMILAR TO
    '[0-9]+ [A-Z][a-z]+ [1-2][0-9][0-9][0-9]'

SELECT *
FROM cable
WHERE content
  SIMILAR TO
    '% [0-9]+ [A-Z][a-z]+ [1-2][0-9][0-9][0-9]%'
```

Outline

- The syntax of POSIX regular expressions is defined with atoms, quantifiers, and constraints



- The syntax of POSIX regular expressions is defined with atoms, quantifiers, and constraints
- As opposed to the previous cases, regex match any part of a string (constraints can be used to match the whole string)

Atoms

`(re)` (where `re` is any regular expression) matches a match for `re`, with the match noted for possible reporting

Atoms

`(re)` (where `re` is any regular expression) matches a match for `re`, with the match noted for possible reporting

`(?:re)` as above, but the match is not noted for reporting (a ”non-capturing” set of parentheses)

Atoms

- `(re)` (where `re` is any regular expression) matches a match for `re`, with the match noted for possible reporting
- `(?:re)` as above, but the match is not noted for reporting (a "non-capturing" set of parentheses)
- `.` matches any single character (similar to _ in `LIKE`)

Atoms

- `(re)` (where `re` is any regular expression) matches a match for `re`, with the match noted for possible reporting
- `(?:re)` as above, but the match is not noted for reporting (a "non-capturing" set of parentheses)
- `.` matches any single character (similar to _ in `LIKE`)
- `[chars]` a bracket expression, matching any one of the chars

Atoms - cont.

`\k` (where `k` is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., \\ matches a backslash character

Atoms - cont.

- `\k` (where `k` is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character
- `\c` where c is alphanumeric (possibly followed by other characters) is an escape

Atoms - cont.

- `\k` (where `k` is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character
- `\c` where c is alphanumeric (possibly followed by other characters) is an escape
- `{` when followed by a character other than a digit, matches the left-brace character `{`; when followed by a digit, it is the beginning of a bound (explained later)

Atoms - cont.

- `\k` (where `k` is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character
- `\c` where c is alphanumeric (possibly followed by other characters) is an escape
- `{` when followed by a character other than a digit, matches the left-brace character `{`; when followed by a digit, it is the beginning of a bound (explained later)
- `x` where x is a single character with no other significance, matches that character

Quantifiers

- a sequence of 0 or more matches of the atom

Quantifiers

- $*$ a sequence of 0 or more matches of the atom
- $+$ a sequence of 1 or more matches of the atom

Quantifiers

- $*$ a sequence of 0 or more matches of the atom
- $+$ a sequence of 1 or more matches of the atom
- ? a sequence of 0 or 1 matches of the atom

Quantifiers

- $*$ a sequence of 0 or more matches of the atom
- $+$ a sequence of 1 or more matches of the atom
- $?$ a sequence of 0 or 1 matches of the atom
- $\{m\}$ a sequence of exactly m matches of the atom

Quantifiers

- `*` a sequence of 0 or more matches of the atom
- `+` a sequence of 1 or more matches of the atom
- `?` a sequence of 0 or 1 matches of the atom
- `{m}` a sequence of exactly `m` matches of the atom
- `{m,}` a sequence of `m` or more matches of the atom

Quantifiers

- `*` a sequence of 0 or more matches of the atom
- `+` a sequence of 1 or more matches of the atom
- `?` a sequence of 0 or 1 matches of the atom
- `{m}` a sequence of exactly `m` matches of the atom
- `{m,}` a sequence of `m` or more matches of the atom
- `{m,n}` a sequence of `m` through `n` (inclusive) matches of the atom; `m` cannot exceed `n`

Constraints

- ^ matches at the beginning of the string
- $ matches at the end of the string

POSIX regular expressions in PostgreSQL

- ~ matches regular expression, case sensitive

POSIX regular expressions in PostgreSQL

- ~ matches regular expression, case sensitive
- ~* matches regular expression, case insensitive

POSIX regular expressions in PostgreSQL

- ~ matches regular expression, case sensitive
- ~* matches regular expression, case insensitive
- !~ does not match regular expression, case sensitive

POSIX regular expressions in PostgreSQL

- ~ matches regular expression, case sensitive
- ~* matches regular expression, case insensitive
- !~ does not match regular expression, case sensitive
- !~* does not match regular expression, case insensitive

```
SELECT
  'Never gonna let you down' ~* '^never'

SELECT
  'Never gonna run around and desert you' !~ '^gonna'

SELECT
  'Never gonna make you cry' ~ 'cry$'

SELECT
  'Never gonna say goodbye' !~* 'SAY$'
```

```
SELECT '27-9-2001' ~ '[0-9]{1,2}\-[0-9]{1,2}\-[1-2][0-9]{3}'

SELECT *
FROM cable
WHERE
  content ~ '[0-9]{1,2}\-[0-9]{1,2}\-[1-2][0-9]{3}'
```

Auxilliary functions

`substring(string from pattern)` where `pattern` is a regular expression extracts the matching substring

Auxilliary functions

`substring(string from pattern)` where `pattern` is a regular expression extracts the matching substring

`regexp_replace(source, pattern, replacement)` replaces the matching substring with `replacement`

Auxilliary functions

`substring(string from pattern)` where `pattern` is a regular expression extracts the matching substring

`regexp_replace(source, pattern, replacement)` replaces the matching substring with `replacement`

`regexp_matches(string, pattern)` returns all matching substrings

Auxilliary functions

`substring(string from pattern)` where `pattern` is a regular expression extracts the matching substring

`regexp_replace(source, pattern, replacement)` replaces the matching substring with `replacement`

`regexp_matches(string, pattern)` returns all matching substrings

`regexp_split_to_table(string, pattern)` uses regular expression match as delimeter and returns splitted table

Auxilliary functions

`substring(string from pattern)` where `pattern` is a regular expression extracts the matching substring

`regexp_replace(source, pattern, replacement)` replaces the matching substring with `replacement`

`regexp_matches(string, pattern)` returns all matching substrings

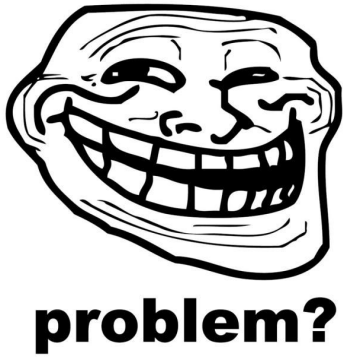
`regexp_split_to_table(string, pattern)` uses regular expression match as delimeter and returns splitted table

`regexp_split_to_array(string, pattern)` same as above, but returns array

```
SELECT substring('Never gonna tell a lie' from 'gonna(.*)lie')

SELECT regexp_replace( '... and hurt you', 'hurt', 'rickroll' )

SELECT regexp_matches( '27 Sep 2001', '([0-9]{1,2})
    ([A-Z][a-z]{2}) ([1-2][0-9]{3})' )

SELECT regexp_split_to_table( 'joza@foi.hr; ivek@foi.hr;
    bara@foi.hr', '\; ' )
```

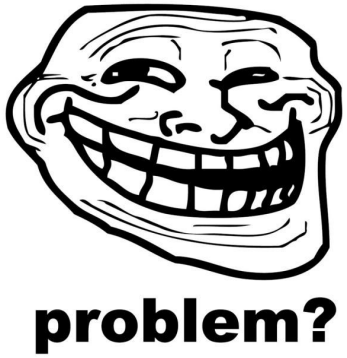
Outline

Full Text Search



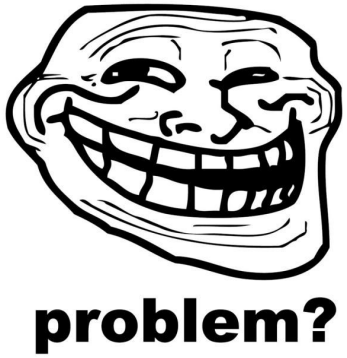
- Problems with pattern matching:

Full Text Search



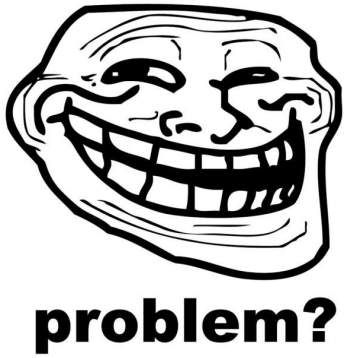
- Problems with pattern matching:
  - No linguistic support (for example verb forms: satisfies, satisfy ...)

Full Text Search



- Problems with pattern matching:
  - No linguistic support (for example verb forms: satisfies, satisfy ...)
  - No ordering/ranking of search results

Full Text Search



- Problems with pattern matching:
  - No linguistic support (for example verb forms: satisfies, satisfy ...)
  - No ordering/ranking of search results
  - Relatively slow due to lack of indexing support

Full text indexing

- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching:

Full text indexing

- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching:
  - **Parsing documents into tokens** - e.g., numbers, words, complex words, email addresses, so that they can be processed differently

Full text indexing

- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching:
  - **Parsing documents into tokens** - e.g., numbers, words, complex words, email addresses, so that they can be processed differently
  - **Converting tokens into lexemes** - normalized strings so that different forms of the same word are made alike

Full text indexing

- Full text indexing allows documents to be preprocessed and an index saved for later rapid searching:
  - **Parsing documents into tokens** - e.g., numbers, words, complex words, email addresses, so that they can be processed differently
  - **Converting tokens into lexemes** - normalized strings so that different forms of the same word are made alike
  - **Storing preprocessed documents optimized for searching** - each document can be represented as a sorted array of normalized lexemes possibly with positional information to use for proximity ranking

Full Text Search in PostgreSQL

PostgreSQL uses the `@@` (match) operator and two additional datatypes:

Full Text Search in PostgreSQL

PostgreSQL uses the `@@` (match) operator and two additional datatypes:

- `tsvector` (preprocessed) text document

Full Text Search in PostgreSQL

PostgreSQL uses the `@@` (match) operator and two additional datatypes:

- `tsvector` (preprocessed) text document
- `tsquery` (preprocessed) query

Full Text Search in PostgreSQL

PostgreSQL uses the `@@` (match) operator and two additional datatypes:

- `tsvector` (preprocessed) text document
- `tsquery` (preprocessed) query

The `@@` operator returns true if a `tsvector` matches a `tsquery`

```
SELECT to_tsvector('a fat cat sat on a mat and ate a fat rat')
    @@ to_tsquery('cat & rat');
 ?column?
----------
 t

SELECT to_tsquery('fat & cow') @@ to_tsvector('a fat cat sat
    on a mat and ate a fat rat');
 ?column?
----------
 f
```

```
SELECT to_tsvector( 'Today is another nice rainy day in
    Manchester!' );
                          to_tsvector
-----------------------------------------------------------
 'anoth':3 'day':6 'manchest':8 'nice':4 'raini':5 'today':1
(1 row)
```

`tsquery` Syntax

The `tsquery` datatype uses the following syntax:

`tsquery` Syntax

The `tsquery` datatype uses the following syntax:

`&` - denotes conjunction (logical AND)

`tsquery` Syntax

The `tsquery` datatype uses the following syntax:

- `&` - denotes conjunction (logical AND)
- `|` - denotes disjunction (logical OR)

`tsquery` Syntax

The `tsquery` datatype uses the following syntax:

- `&` - denotes conjunction (logical AND)
- `|` - denotes disjunction (logical OR)
- `!` - denotes negation (logical NOT)

`tsquery` Syntax

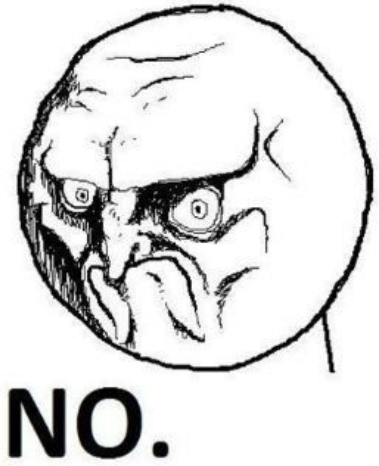
The `tsquery` datatype uses the following syntax:

- `&` - denotes conjunction (logical AND)
- `|` - denotes disjunction (logical OR)
- `!` - denotes negation (logical NOT)
- `()` - bracket are used to group expressions together

```
SELECT
 to_tsvector( 'E-discovery discovers new knowledge in huge
    haystacks of data' )
 @@
 to_tsquery( 'new & ( ! huge | data )' );
```

YAY!

```
SELECT
 to_tsvector( 'E-discovery discovers new knowledge in huge
    haystacks of data' )
 @@
 to_tsquery( 'new & ! ( huge | data )' )
```


NO.

```
SELECT *
FROM
  cable c,
  to_tsquery( 'siemens & ( iran | iraq ) & ! kuwait' ) q
WHERE q @@ to_tsvector( content )
```

Ranking Search Results

- PostgreSQL has currently implemented two ranking functions:

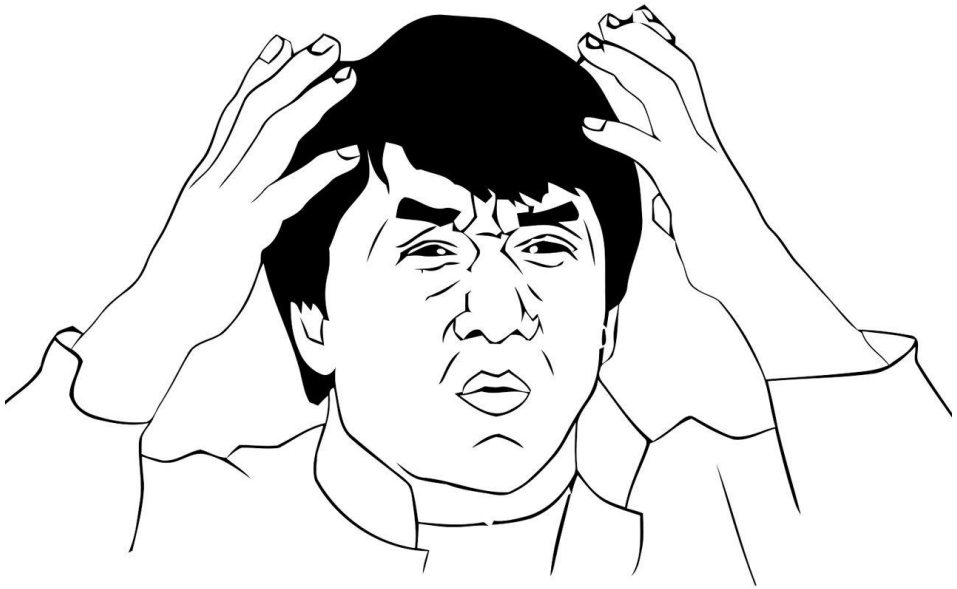
Ranking Search Results

- PostgreSQL has currently implemented two ranking functions:
  - `ts_rank( vector, query )` - based on frequency of matching lexems

Ranking Search Results

- PostgreSQL has currently implemented two ranking functions:
  - `ts_rank( vector, query )` - based on frequency of matching lexems
  - `ts_rank_cd( vector, query )` - based on cover density ranking

```
SELECT c.id, ts_rank_cd( p.to_tsvector, q ) AS rank
FROM
  cable c,
  preprocessed p,
  to_tsquery( 'siemens & ( iran | iraq )' ) q
WHERE
  c.id = p.id
  AND q @@ p.to_tsvector
ORDER BY rank DESC
LIMIT 5
   id    |    rank
--------+-----------
 230290 |  0.160565
 184343 | 0.0525963
 241208 | 0.0509475
 232601 | 0.0353333
 201137 | 0.0302077
(5 rows)
```



Outline

Aggregate functions

`avg` finds the average of a numerical attribute

Aggregate functions

`avg` finds the average of a numerical attribute

`sum` finds the sum of a numerical attribute

Aggregate functions

`avg` finds the average of a numerical attribute

`sum` finds the sum of a numerical attribute

`min` find the smalest value of an attribute

Aggregate functions

- `avg` finds the average of a numerical attribute
- `sum` finds the sum of a numerical attribute
- `min` find the smalest value of an attribute
- `max` find the highest value of an attribute

Aggregate functions

- `avg` finds the average of a numerical attribute
- `sum` finds the sum of a numerical attribute
- `min` find the smalest value of an attribute
- `max` find the highest value of an attribute
- `count` find the number of values of some attribute

Aggregate functions

- `avg` finds the average of a numerical attribute
- `sum` finds the sum of a numerical attribute
- `min` find the smalest value of an attribute
- `max` find the highest value of an attribute
- `count` find the number of values of some attribute
- `variance` find the variance of some numerical attribute

Aggregate functions

- `avg` finds the average of a numerical attribute
- `sum` finds the sum of a numerical attribute
- `min` find the smalest value of an attribute
- `max` find the highest value of an attribute
- `count` find the number of values of some attribute
- `variance` find the variance of some numerical attribute
- `stddev` find the standard deviation of some numerical attribute

```
SELECT count(*)
FROM cable

SELECT min(date), max(date)
FROM cable
```

GROUP BY and HAVING

Aggregate functions are usually used with a GROUP BY clause to group aggregate values by some given attribute.

GROUP BY and HAVING

Aggregate functions are usually used with a GROUP BY clause to group aggregate values by some given attribute.

HAVING is used for constraints on aggragate expressions.

```
SELECT date, count(*)
FROM cable
GROUP BY date
ORDER BY 2 DESC
```

```
SELECT origin, count(*)
FROM cable
GROUP BY 1
HAVING count(*) > 100
ORDER BY 2
```

Gathering Text Search Statistics

- PostgreSQL provides you also with the `ts_stat( sql_query )` function which allows you to gather useful statistics

Gathering Text Search Statistics

- PostgreSQL provides you also with the `ts_stat( sql_query )` function which allows you to gather useful statistics
- Note that `ts_stat` returns a set of records of the form `( word, ndoc, nentry )`:

Gathering Text Search Statistics

- PostgreSQL provides you also with the `ts_stat( sql_query )` function which allows you to gather useful statistics
- Note that `ts_stat` returns a set of records of the form `( word, ndoc, nentry )`:

  `word` - the lexem (word)

Gathering Text Search Statistics

- PostgreSQL provides you also with the `ts_stat( sql_query )` function which allows you to gather useful statistics
- Note that `ts_stat` returns a set of records of the form `( word, ndoc, nentry )`:

  `word` - the lexem (word)
  `ndoc` - the number of documents the lexem occurs in

Gathering Text Search Statistics

- PostgreSQL provides you also with the `ts_stat( sql_query )` function which allows you to gather useful statistics
- Note that `ts_stat` returns a set of records of the form `( word, ndoc, nentry )`:

  `word` - the lexem (word)
  `ndoc` - the number of documents the lexem occurs in
  `nentry` - the total number of occurences of the lexem

```
SELECT *
FROM ts_stat( 'SELECT to_tsvector FROM preprocessed' )
ORDER BY
  nentry DESC,
  ndoc DESC,
  word
LIMIT 10;
```

Outline

Mini assignment

Keyword/pattern definition

Define a set of keywords and data possibly found in cables which for you as an investigator would be of importance. What kind of (structured) data is important to an investigative process? Which kind of statistics about these data will indicate if they are interesting or not?

Mini assignment

Keyword/pattern definition

Define a set of keywords and data possibly found in cables which for you as an investigator would be of importance. What kind of (structured) data is important to an investigative process? Which kind of statistics about these data will indicate if they are interesting or not?

- e.g. keywords: risk, plan, transaction, signature, intelligence etc.

Mini assignment

Keyword/pattern definition

Define a set of keywords and data possibly found in cables which for you as an investigator would be of importance. What kind of (structured) data is important to an investigative process? Which kind of statistics about these data will indicate if they are interesting or not?

- e.g. keywords: risk, plan, transaction, signature, intelligence etc.
- e.g. data: money amounts, telephone numbers, email addresses, snailmail addresses, domains, dates, account numbers, etc.

Mini assignment

Keyword/pattern definition

Define a set of keywords and data possibly found in cables which for you as an investigator would be of importance. What kind of (structured) data is important to an investigative process? Which kind of statistics about these data will indicate if they are interesting or not?

- e.g. keywords: risk, plan, transaction, signature, intelligence etc.
- e.g. data: money amounts, telephone numbers, email addresses, snailmail addresses, domains, dates, account numbers, etc.

Try to create patterns and queries that will allow you to search for these data!

Mini assignment

Keyword/pattern definition

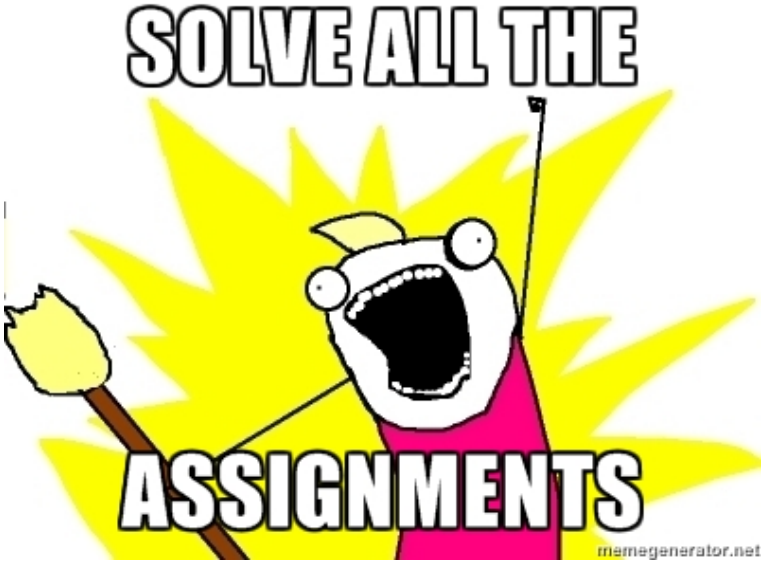
Define a set of keywords and data possibly found in cables which for you as an investigator would be of importance. What kind of (structured) data is important to an investigative process? Which kind of statistics about these data will indicate if they are interesting or not?

- e.g. keywords: risk, plan, transaction, signature, intelligence etc.
- e.g. data: money amounts, telephone numbers, email addresses, snailmail addresses, domains, dates, account numbers, etc.

Try to create patterns and queries that will allow you to search for these data!

Database search application

Create a small application that will allow you to search the database and gather statistics. Think about your user friendlyness!


SOLVE ALL THE
ASSIGNMENTS
memegenerator.net