JUGTAAS - **JavaDay** 2017

007: Agent Under Fire
to
Ω-logic

fun with ETL and full text search (and Java and Postgres)

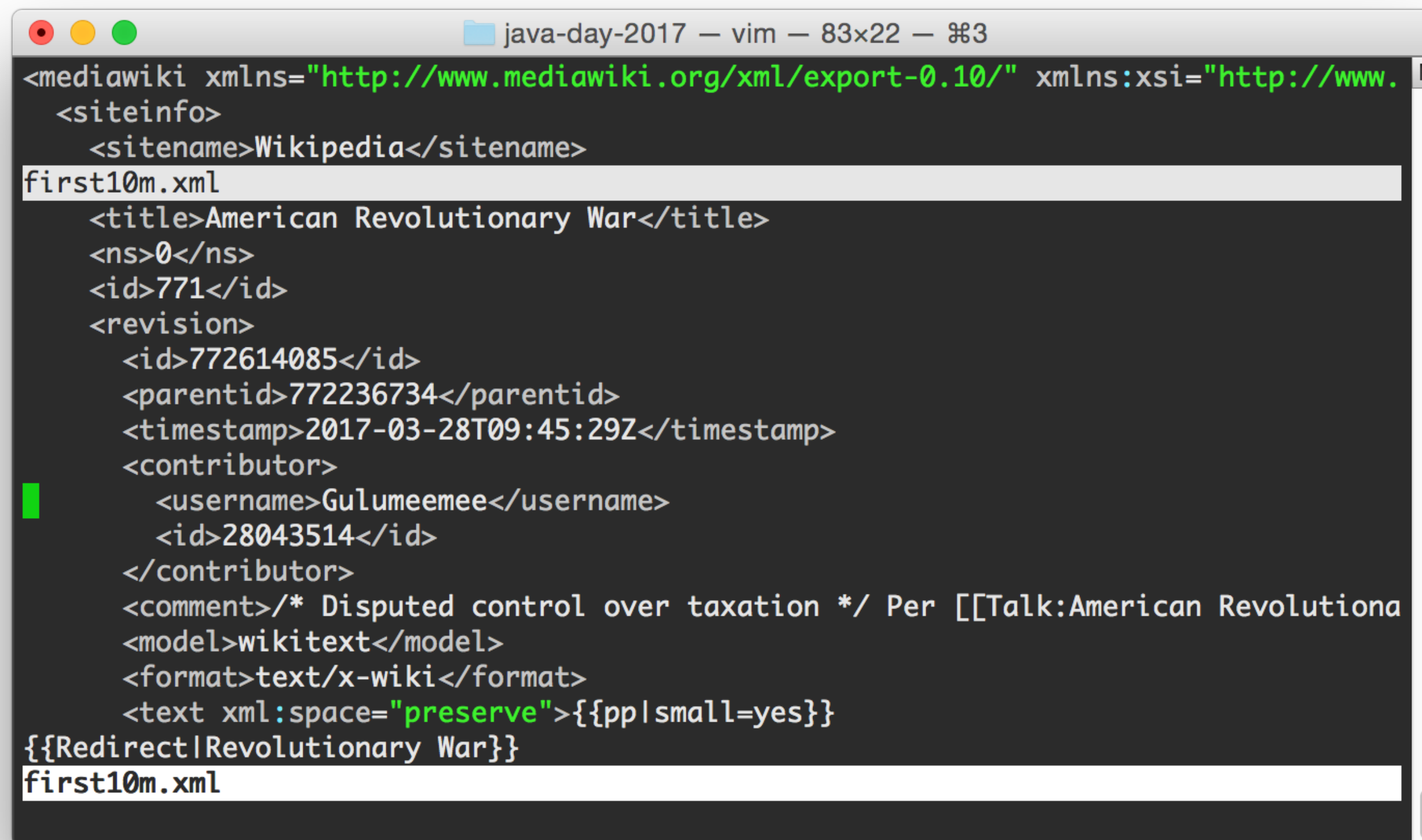
Chris Mair - http://www.1006.org

step1: get something big

- one can actually download Wikipedia ;)
- https://dumps.wikimedia.org/enwiki/20170401/
- let's get the current wikipedia in English (enwiki-20170401-pages-articles.xml.bz2)
- the multistream version is the same thing but suitable for parallel bzip2 implementations

bigish?

- XML: 13GB compressed, 58G uncompressed

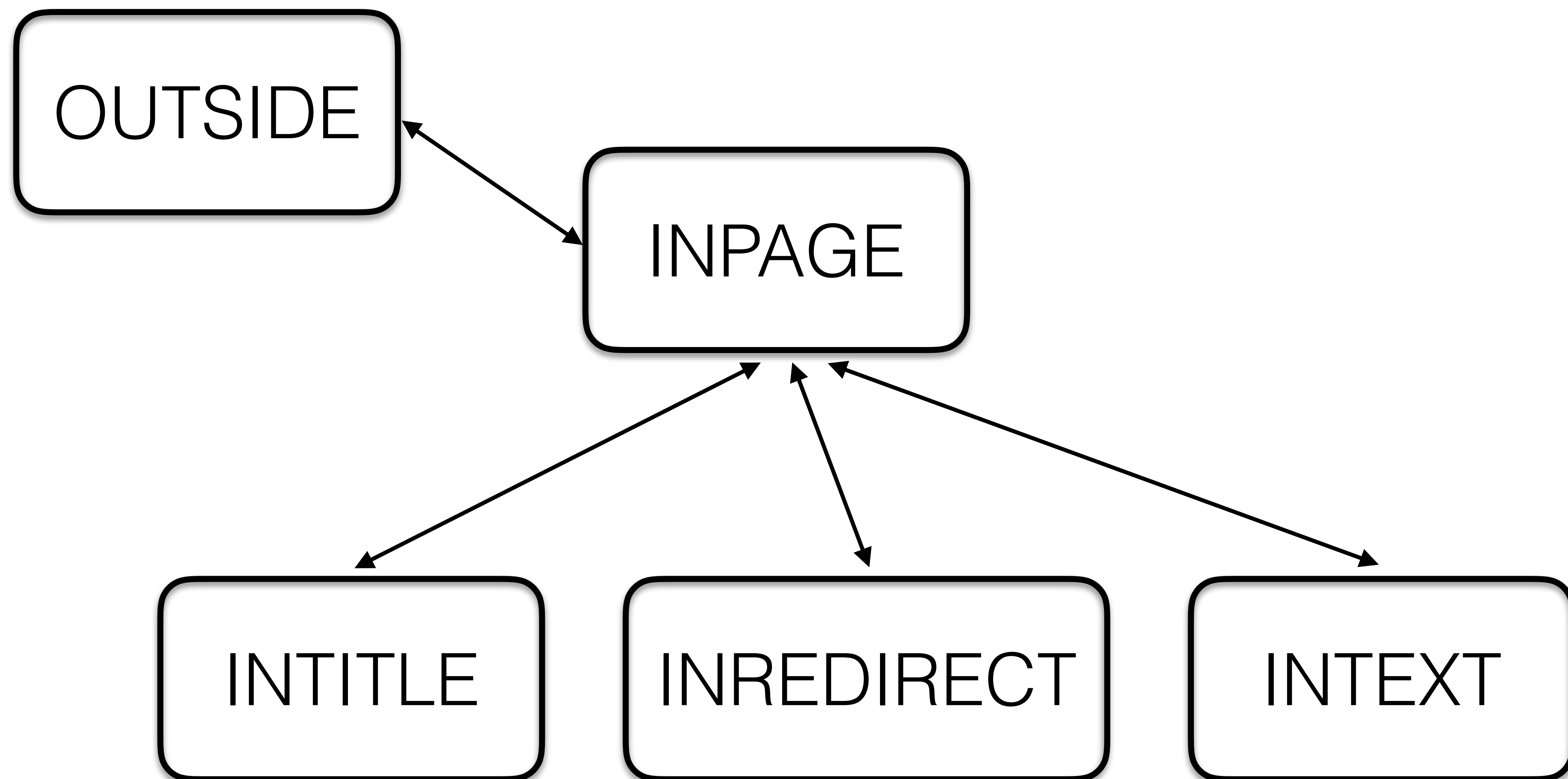


don't try opening the whole thing in vim ;)

how to parse this?

- DOM is a bad idea, unless you have really really much RAM ;)
- need to use SAX or StAX (Streaming API for XML)
- I picked StAX: this allows running a cursor through the document pulling piece by piece
- we don't need everything: just titles and article texts (but not meta articles)

a small state machine



it's easier in code...

```
while (r.hasNext()) {
    int e = r.next();
    switch (state) {

        case OUTSIDE:
            if (startElement(e, r, "page")) {
                state = State.INPAGE;
            }
            break;

        case INPAGE:
            if (startElement(e, r, "title")) {
                state = State.INTITLE;
            } else if (startElement(e, r, "redirect")) {
                isRedirect = true;
                state = State.INREDIRECT;
            } else if (startElement(e, r, "text")) {
                state = State.INTEXT;
            } else if (endElement(e, r, "page")) {
```

have a look at WikiETL.java...

at the end of a page...

- we have the title and the article (text under <text>)
- we can skip redirections
- we can skip meta pages (title.startsWith("Category:"), title.startsWith("File:") and the like)
- let's also skip disambiguation pages (title.endsWith("(disambiguation)"))

it's true what they say...

- first I used String article and += to assemble the pieces under <text>...
- this is slooow!
- replacing with StringBuilder and append() and setLength(0) really helped
- ~ 20 times faster for this use case!

running into obscure limits

```
[chris@moon mnt]$ java -jar WikiETL.jar > out

javax.xml.stream.XMLStreamException: ParseError at [row,col]:[64151498,1332]

Message: JAXP00010004: The accumulated size of entities is "50,000,001" that
exceeded the "50,000,000" limit set by "FEATURE_SECURE_PROCESSING".

    at com.sun.org.apache.xerces.internal.impl.XMLStreamReaderImpl.next
   (XMLStreamReaderImpl.java:604)
    at wikietl.WikiETL.main(WikiETL.java:37)
```

- if you google this, you'll find the internet is full of people that parse Wikipedia as a hobby ;)
- solution: -Djdk.xml.totalEntitySizeLimit=0

store this into postgres

```
create table wiki (
    id      serial primary key,
    title   varchar,
    article varchar
);
```

- use a prepared statement!
- setAutoCommit(false);
- commit() after every 1k articles stored

success!

```
5328170 articles imported in 2029 seconds

real     33m50.065s
user     11m43.420s
sys      2m18.632s
```

wait a sec!
Java used only 1/3 of the time!?

- Java used at most 150 MB resident set size (OpenJDK 8 under Linux kernel 4.9)
- this is machine with 2 vCPUs (Xeon E5 v4 at 2.3 GHz), 15GB RAM and a local SSD on AWS ("i3.large" = the *smallest* of 10 in the storage optimized category)

what's Postgres doing?

- by default, Postgres *compresses* column values that are larger than 8KB - most articles are
- this is known as TOAST (**T**he **O**versized-**A**ttribute **S**torage **T**echnique)
- Postgres does this since version 7.1 released in 2001, but many people are not aware of this ;)
- for our use case - storing large texts in a storage optimized VM with a fast local disk - this isn't ideal!

you can control TOAST!

```
alter table wiki
  alter article set storage external;
```

- EXTENDED allows both out-of-line storage (TOAST) and compression - this is the default
- EXTERNAL allows out-of-line storage (TOAST), but not compression!
- note you need to run the alter statement *before* populating the table (!)

ta-dah!

```
5328170 articles imported in 1392 seconds

real    23m12.445s
user    11m36.732s
sys     2m19.332s
```

OK, now it's ~ half and half between Java and Postgres

- we gained **30%** time at the cost of space:

```
wiki=# \dt+
                    List of relations
 Schema |    Name     | Type  | Owner | Size
--------+-------------+-------+-------+-------
 public | wiki        | table | chris | 36 GB
 public | wikinormal  | table | chris | 19 GB
(2 rows)
```

uncompressed

normal behaviour: compressed!

no compression overhead

- now all table scans on this setup will be about ~ 100ms faster per ~ 1000 articles:

```
wiki=# select sum(length(article)) from wikinormal where id < 1000;
   sum
----------
 37679901
(1 row)

Time: 217.491 ms

wiki=# select sum(length(article)) from wiki where id < 1000;
   sum
----------
 37679901
(1 row)

Time: 115.994 ms
```

yeah, but:
what about full text search?

```
wiki=# select title from wiki where id < 1000 and article ilike '%Bolzano%';
         title
-----------------------
 Augustin-Louis Cauchy
(1 row)

Time: 527.777 ms
```

- we need some sort of indexing, otherwise this doesn't scale to millions of documents
- we need to have linguistic support to handle derived words (e.g. *satisfy* is the same thing as *satisfies* for searching)
- we need to handle stop word (e.g. the, is, an, ...)
- we need some sort of ranking

full text search is built-in!

- to_tsvector() tokenizes a natural language string into lexemens and eliminates stop words!

```
chris=# select to_tsvector('english', 'this is an interesting database system.');
             to_tsvector
-------------------------------------
 'databas':5 'interest':4 'system':6
```

this is a **tsvector**: a data type that can hold word stems and their position(s) in the text

```
chris=# select to_tsvector('german',
        'ich hätte gerne Frühstück!');
           to_tsvector
---------------------------------
 'fruhstuck':4 'gern':3 'hatt':2
```

```
chris=# select to_tsvector('italian', 'siamo alla facoltà di sociologia.');
       to_tsvector
-------------------------
 'facolt':3 'sociolog':5
```

full text search is built-in!

- to_tsquery() does the same thing for a search

```
chris=# select to_tsquery('italian', 'facoltà & sociologia');
      to_tsquery
-----------------------
 'facolt' & 'sociolog'         -> contains facoltà AND sociologia

chris=# select to_tsquery('italian', 'facoltà | sociologia');
      to_tsquery
-----------------------
 'facolt' | 'sociolog'         -> contains facoltà OR sociologia

chris=# select to_tsquery('italian', 'facoltà <-> sociologia');
       to_tsquery
-------------------------
 'facolt' <-> 'sociolog'       -> contains facoltà FOLLOWED BY sociologia

chris=# select to_tsquery('italian', 'facoltà & !sociologia');
       to_tsquery
------------------------
 'facolt' & !'sociolog'        -> contains facoltà, BUT NOT sociologia
```

full text search is built-in!

- the @@ operator matches a ts_query to a ts_vector

```
chris=# select to_tsvector('italian', 'siamo alla facoltà di fisica') @@
               to_tsquery('italian', 'facoltà & ! sociologia');

 ?column?
----------
 t
(1 row)


wiki=# select title from wiki
         where to_tsvector('english', article) @@
               to_tsquery('english', 'squirrel') and id < 100;

  title
---------
 Alberta
(1 row)

Time: 936.698 ms
```

1 sec for 100 articles, that's crazy slow!?

Postgres can index everything

- thanks to GIN (Generalized Inverted Index) support you can index vectors, such as the ts_vector

```
wiki=# CREATE INDEX wiki100ix ON wiki
       USING gin (to_tsvector('english', article))
       WHERE id < 100;

CREATE INDEX
Time: 1208.925 ms
```

OK, this very cool index is - all at the same time:

- a **generalized inverted index**,
  because it works on vector values
- a **partial index**,
  because I'm trying it out on the first 100 rows only
- a **functional index**,
  because it's indexing a function, not a column

let's look for the squirrel

- the planner will automatically use the matching index (and of course keep it up to date too):

```
wiki=# select title from wiki
         where to_tsvector('english', article) @@
               to_tsquery('english', 'squirrel') and id < 100;

  title
---------
 Alberta
(1 row)

Time: 0.571 ms
```

now we're talking!
that's ~ 2000 time faster

Ok, but that just 100 articles

- sure, here is the **full index**! it took 3 hours, 21 minutes and 40 seconds to create it: that's ~ 440 articles per second indexed

```
wiki=# \di+
                                   List of relations
 Schema |       Name        | Type  | Owner |    Table     |  Size
--------+-------------------+-------+-------+--------------+--------
 public | wiki_pkey         | index | chris | wiki         | 114 MB
 public | wikinormal_pkey   | index | chris | wikinormal   | 114 MB
 public | wikiall           | index | chris | wiki         | 10 GB
```

```
wiki=# \dt+
                          List of relations
 Schema |    Name    | Type  | Owner | Size
--------+------------+-------+-------+-------
 public | wiki       | table | chris | 36 GB
 public | wikinormal | table | chris | 19 GB
```

needle and haystack solved

- find the 9135 articles out of 5328170 mentioning squirrels in 12 ms or find the 4 articles mentioning a squirrel, a spoon, a bridge and a cactus in 3 ms:

```
wiki=#  select count(id) from wiki
        where to_tsvector('english', article) @@ to_tsquery('english', 'squirrel');
 count
-------
  9135

Time: 12.193 ms

wiki=#  select id, title from wiki
         where to_tsvector('english', article) @@
               to_tsquery('english', 'squirrel & cactus & bridge & spoon');
   id    |                    title
---------+----------------------------------------------
   24819 | Phoenix, Arizona
  934792 | Index of Arizona-related articles
 4974798 | List of Hi-5 episodes
 5190035 | List of Encyclopædia Britannica Films titles

Time: 3.155 ms
```

ranking squirrels

- there are 1481 articles mentioning **squirrels** and **Washington** and we can quickly find them all:

```
wiki=# select count(*)
  from wiki
    where to_tsvector('english', article) @@
          to_tsquery('english', 'squirrel & washington');

 count
-------
  1481
(1 row)

Time: 5.618 ms
```

ranking squirrels

- can we rank these articles somehow?

```
select
    ts_rank_cd(to_tsvector('english', article),
               to_tsquery('english', 'squirrel & washington')
              ) as position,
    title
from wiki
    where to_tsvector('english', article) @@
          to_tsquery('english', 'squirrel & washington')
    order by position desc limit 15;
```

ranking squirrels

- This works well, but is slow!

```
 position |                        title
----------+------------------------------------------------------
 0.56301  | Washington ground squirrel
 0.118487 | Black squirrel
 0.115948 | Foster Coulee
 0.103692 | List of mammals described in the 21st century
 0.103236 | Northern goshawk
 0.100733 | Yup'ik clothing
 0.100719 | List of deaths in rock and roll
 0.100667 | Virginia
 0.100645 | List of train songs
 0.100546 | Great horned owl
 0.100334 | Flying Hawk
 0.100276 | List of Encyclopædia Britannica Films titles
 0.100186 | Mount Everest
  0.10015 | List of shipwrecks in 1825
 0.100101 | List of Ripley's Believe It or Not! episodes (1982–86)
(15 rows)

Time: 13634.014 ms
```

ranking squirrels

- It is slow because the first argument to ts_rank_cd() is a to_tsvector() that has to parse the 1481 articles again...
- the recommended solution is to add a column with the precomputed vector to_tsvector('english', article)!
- this way I can index this new column instead of using a functional index, but I need a trigger to keep this up to date
- I didn't have time to prepare a demo for this :)

'k thx bye ;)

(C) Chris Mair 2017 - License: Creative Commons BY-SA.