solve this problem [5/5]

Andrea Giugliano

January 14, 2018

Contents

- CLOSING NOTE *[2017-01-22 Sun 20:35]*

See: Re: Application to Clojure Developer role

1 the exercise

Get the file:

```
cd /tmp
wget http://exchange.somecoolcompany.com.s3-website.eu-central-1.amazonaws.com/somecool
```

Untar the main file:

```
tar xvf somecoolcompany-backend-test.tar

# Somecoolcompany backend developer test

## The task

There is a PostgreSQL table of persons (person), uniquely identified
by their first name (fname), last name (lname) and date of birth
(dob).  Every person has a telephone number (phone).

This table needs to be updated from an XML file containing elements of
the form

‘‘‘
<member>
 <first-name>JOHN</first-name>
 <last-name>DOE</last-name>
 <date-of-birth>2002-02-01</date-of-birth>
 <phone>9548938821</phone>
</member>
‘‘‘
```

```
If the phone number is already correct, nothing should be changed in
the database.  If a person record does not exist, it needs to be
created.

The person database table contains 10 million rows.

The update file contains 1.5 million entries.

## Objective

 - Write clean code that performs the operation correctly
 - Provide basic loading statistics at the end of the operation
 - Use proper mechanisms to process the input file
 - Find ways to minimize the overall run time of the merge process
 - Reason about performance and memory usage

The number of records in the sample database and the input file are
meant to reflect the number of records in a production system.  A
production system would have more individual fields per person,
consider that when choosing an implementation strategy.

## Files

The file person.sql.gz contains a database dump of the person table
which can be imported into PostgreSQL.

The file update-file.xml.gz contains the XML input file to be merged
into the database.
```

Here Re: Application to Clojure Developer role Volker says that I can use open source libraries, and no deadline. Also about this:

> The statistics should be useful to the user of the software and relate to the input business objects, not the database system.

I think they want to know just how many attributes were updated, how many were present already, and how many were created.

2 DONE the clojure setup

- CLOSING NOTE *[2017-01-18 Wed 17:08]*
- Note taken on *[2017-01-18 Wed 09:57]*
  `https://clojuredocs.org/clojure.xml/parse`
- Note taken on *[2017-01-18 Wed 09:56]*
  clj+postgres: `https://github.com/alaisi/postgres.async`

We create a new clojure project for the somecoolcompany-exercise:

```
lein new trysomecoolcompany-exercise
```

Then we modify the project metadata to include the somecoolcompany-exercise library:

```
 (defproject trysomecoolcompany-exercise "0.1.0-SNAPSHOT"
     :description "somecoolcompany-exercise"
:url "http://example.com/FIXME"
:license {:name "Eclipse Public License"
:url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[org.clojure/clojure "1.8.0"]
               [org.clojure/data.zip "0.1.2"]
               [org.clojure/data.xml "0.2.0-alpha2"]
               [alaisi/postgres.async "0.8.0"]]
   :main trysomecoolcompany-exercise.core)
```

Then we install the dependencies:

```
lein deps
```

3 DONE the db setup

`http://stackoverflow.com/questions/6842393/import-sql-dump-into-postgresql-database`
`https://www.postgresql.org/docs/current/static/role-attributes.html`

Now let's work with the database.

Unzip sql file:

```
gunzip -q person.sql.gz
```

I need to install and initialize postgres:

```
nix -iA nixos.postgresql
```

Then I have to initialize it, so let's create a directory for this:

```
mkdir db
initdb -D db
pg_ctl -D db -l logfile start
# we need to create the user hans
echo "CREATE ROLE hans LOGIN;"> patch.sql
# we need to alter the table to have large text fields
psql postgres < patch.sql
```

finally we import the provided sql dump after patching the NULL string with the dummy value 5500-12-12 (there are few NULL strings in the data values and they would make the primary key constraint invalid):

```
sed -i "s/\\\N/5500-12-12/g"  /tmp/somecoolcompany-backend-test/person.sql
psql postgres < /tmp/somecoolcompany-backend-test/person.sql
```

finally we add a primary key, or better unique indexes to avoid the NULL problem:

```
echo "ALTER TABLE person ADD PRIMARY KEY (fname, lname, dob);" > patch2.sql
#echo "CREATE UNIQUE INDEX p_col_uni_idx ON person (fname, lname, dob) WHERE fname IS N
#echo "CREATE UNIQUE INDEX p_col_a_uni_idx ON person (fname, lname) WHERE dob IS NULL;"
#echo "CREATE UNIQUE INDEX p_col_b_uni_idx ON person (fname, dob) WHERE lname IS NULL;"
psql postgres < patch2.sql
```

4 DONE the xml setup

- CLOSING NOTE *[2017-01-18 Wed 18:18]*
  I moved to data.xml to have lazy reading of the big xml file
- CLOSING NOTE *[2017-01-18 Wed 11:42]*

Unzip xml file:

```
gunzip -q update-file.xml.gz
```

`http://stackoverflow.com/questions/11537923/searching-xml-in-clojure` there is also an xpath library, but maybe I do not need to query the file so much: `https://github.com/kyleburton/clj-xpath/`

Let's test a bit this zip library.

4.1 DONE let's run the stackoverflow example as it is

- CLOSING NOTE *[2017-01-18 Wed 11:18]*

`http://stackoverflow.com/questions/11537923/searching-xml-in-clojure`
The xml file:

```
<data>
  <products>
    <product>
      <section>Red Section</section>
      <images>
        <image>img.jpg</image>
        <image>img2.jpg</image>
      </images>
    </product>
    <product>
      <section>Blue Section</section>
      <images>
        <image>img.jpg</image>
        <image>img3.jpg</image>
      </images>
    </product>
    <product>
      <section>Green Section</section>
      <images>
        <image>img.jpg</image>
        <image>img2.jpg</image>
      </images>
    </product>
  </products>
</data>
```

```
(ns core
  (:use clojure.data.zip.xml)
  (:require [clojure.zip :as zip]
            [clojure.xml :as xml]))


(def data (zip/xml-zip (xml/parse "/tmp/test-xml.xml")))
(def products (xml-> data :products :product))
```

```
(for [product products :let [image (xml-> product :images :image)]
                       :when (some (text= "img2.jpg") image)]
  {:section (xml1-> product :section text)
   :images (map text image)})
```

4.2 DONE now let's play a bit to understand the basics

- CLOSING NOTE *[2017-01-18 Wed 11:39]*
  cool!

```
(ns core
  (:use clojure.data.zip.xml)
  (:require [clojure.zip :as zip]
            [clojure.xml :as xml]))


(def data (zip/xml-zip (xml/parse "/tmp/test-xml.xml")))
(def products (xml-> data :products :product))

(for [product products :let [image (xml-> product :images :image)]
                       ; let's filter the Blue Section only
                       :when (some (text= "Blue Section") (xml-> product :section))] ;
  {:section (xml1-> product :section text)
   :images (map text image)})
```

4.3 DONE let's understand this better

- CLOSING NOTE *[2017-01-18 Wed 11:42]*

`https://ravi.pckl.me/short/functional-xml-editing-using-zippers-in-clojure/`

So, now I recollect: we use the zipper data structure that was firstly published in a Haskell paper. Basically is an iterator that maintain its context so can iterate in whatever dimension.

The `xml->` and `xml1->` differ in the number of results returned.

4.4 DONE let's use data.xml to have a lazy reading of xml

- CLOSING NOTE *[2017-01-18 Wed 18:18]*

since we have a huge xml, we cannot have it all as a string.

clojure.data.xml to the rescue: it imports an element at a time from a reader.

However I should still be able to use the zipper to query it:

```
(ns core
  (:use clojure.data.zip.xml)
  (:require [clojure.zip :as zip]
            [clojure.xml :as xml]
            [clojure.data.xml :as data.xml]))

(def data (zip/xml-zip (data.xml/parse (clojure.java.io/input-stream "/tmp/test-xml.xml

(def products (xml-> data :products :product))

(for [product products :let [image (xml-> product :images :image)]
                       :when (some (text= "img2.jpg") image)]
  {:section (xml1-> product :section text)
   :images (map text image)})
```

[org.clojure/data.xml "0.0.8"]

5 DONE test environment [4/4]

- CLOSING NOTE *[2017-01-18 Wed 17:00]*

5.1 DONE create a test db

- CLOSING NOTE *[2017-01-18 Wed 16:12]*

Okay, I want to create a new table called `test_person` with some data in it:

```
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
```

```
SET client_min_messages = warning;

--
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;


--
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
--

COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';


SET search_path = public, pg_catalog;

SET default_tablespace = '';

SET default_with_oids = false;

--
-- Name: person; Type: TABLE; Schema: public; Owner: hans; Tablespace:
--

CREATE TABLE test_person (
    fname character varying,
    lname character varying,
    dob date,
    phone character(10),
    PRIMARY KEY(fname, lname, dob) -- we need a primary key on the fields
);


ALTER TABLE person OWNER TO hans;
--
-- Data for Name: test_person; Type: TABLE DATA; Schema: public; Owner: hans
--
```

```
COPY test_person (fname, lname, dob, phone) FROM stdin;
JIARA HERTZEL 1935-06-05 5859012134
RONJARVIOU COMELLO 1932-09-27 7702713416
\.

--
-- Name: public; Type: ACL; Schema: -; Owner: hans
--

REVOKE ALL ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM hans;
GRANT ALL ON SCHEMA public TO hans;
GRANT ALL ON SCHEMA public TO PUBLIC;



--
-- PostgreSQL database dump complete
--
```

and we import it in the database:

```
psql postgres < /tmp/trysomecoolcompany-exercise/resources/test_person.sql
```

5.2 DONE try postgres-async

- CLOSING NOTE *[2017-01-18 Wed 16:37]*

```
(require '[clojure.core.async :as async :refer [<! >! <!! timeout chan alt! go]])
(require '[postgres.async :refer :all])

(def db (open-db {:hostname "localhost"
                  :port 5432 ; default
                  :database "postgres"
                  :username "andrea"
                  :password ""
                  :pool-size 25})) ; default

;; let's get attributes of the test table
(<!! (query! db ["select * from test_person"]))

;; let's insert a test record
```

```
(<!! (insert! db {:table "test_person"} {:fname "Fiona", :lname "Lullaby", :dob "2010-1

;; let's get the test row
(<!! (query! db ["select * from test_person where fname='Fiona'"]))

;; finally let's remove the test record
(<!! (execute! db ["delete from test_person where fname='Fiona' and lname='Lullaby' and

;; now we cannot find the test row anymore
(<!! (query! db ["select * from test_person where fname='Fiona'"]))

(close-db! db)
```

5.3 DONE create a test xml

- CLOSING NOTE *[2017-01-18 Wed 09:55]*

```
<members>
 <member>
  <firstname>JOHN</firstname>
  <lastname>DOE</lastname>
  <date-of-birth>2002-02-01</date-of-birth>
  <phone>9548938821</phone>
 </member>
 <member>
  <firstname>Fiona</firstname>
  <lastname>Lullaby</lastname>
  <date-of-birth>2010-02-01</date-of-birth>
  <phone>9548938822</phone>
 </member>
 <member>
  <firstname>XX</firstname>
  <lastname>YY</lastname>
  <date-of-birth>2000-02-01</date-of-birth>
  <phone>9548138821</phone>
 </member>
</members>
```

5.3.1 DONE now we can test :)

- CLOSING NOTE *[2017-01-18 Wed 17:00]*

- Note taken on *[2017-01-18 Wed 11:37]*
  to import the xml resource we can use

```
(ns core
  (:use clojure.data.zip.xml)
  (:require [clojure.zip :as zip]
            [clojure.xml :as xml]))


(def test-xml
  (-> "test.xml"
      clojure.java.io/resource
      clojure.java.io/input-stream
      xml/parse
      zip/xml-zip))


;; let's get the name for fun
(xml-> test-xml :members :member :first-name text)
```

```
nil#'core/test-xml("JOHN")
```

6 DONE finally some code (by using Test Driven Development) [9/9]

- CLOSING NOTE *[2017-01-22 Sun 20:27]*

6.1 DONE check in the given sql file how they add records

- CLOSING NOTE *[2017-01-18 Wed 17:04]*
  no, they do not set any primary key on the table person
- Note taken on *[2017-01-18 Wed 11:11]*
  fname,sname,date is the key tuple: they should add this constraint somehow

6.2 DONE feature 0: create a clj representation of person record from xml

- Note taken on *[2017-01-18 Wed 11:51]*
  test that the representation can be converted back to the same xml

- keep in mind that the DB could own more fields

  > A production system would have more individual fields per person, consider that when choosing an implementation strategy.

First tests

```
(ns trysomecoolcompany-exercise.core-test
  (:require [clojure.test :refer :all]
            [trysomecoolcompany-exercise.core :refer :all]))

(def url
  (-> "test.xml"
      clojure.java.io/resource
      str))

(deftest xml->clj-vector-maps-test
  (testing "xml->clj should produce a list of maps with keys :fname :lname :dob :phone"
    (let [expected (list {:fname "JOHN"
                          :lname "DOE"
                          :dob "2002-02-01"
                          :phone "9548938821"
                         }
                         {:fname "Fiona"
                          :lname "Lullaby"
                          :dob "2010-02-01"
                          :phone "9548938822"
                         }
                         {:fname "XX"
                          :lname "YY"
                          :dob "2000-02-01"
                          :phone "9548138821"
                         })]
    (is (= (xml->clj url) expected)))))

(deftest xml->clj-keys-test
  (testing "xml->clj->db should contain only :fname :lname :dob :phone"
    (is (= (set (mapcat keys (xml->clj url))) #{:fname :lname :dob :phone}))))
```

Now code to pass them:

```
(ns trysomecoolcompany-exercise.core
  (:require [clojure.zip :as zip]
            [clojure.data.zip.xml :refer :all]
            [clojure.data.xml :as data.xml]))

(defn get-zipper
  "creates a zipper from an xml url"
  [url]
  (-> url
      clojure.java.io/input-stream
      data.xml/parse
      zip/xml-zip))

(defn xml->clj
  "transforms zipper nodes in maps"
  [url]
  (let [zxml (get-zipper url)
        members (xml-> zxml :members :member)]
    (for [member members  :let [fname (xml1-> member :first-name text)
                                lname (xml1-> member :last-name text)
                                dob (xml1-> member :date-of-birth text)
                                phone (xml1-> member :phone text)]
          ]
  {:fname fname :lname lname :dob dob :phone phone})))
```

6.3 DONE feature 1: create a person record from xml

- CLOSING NOTE *[2017-01-19 Thu 00:42]*

Let's add the database now:

```
(ns trysomecoolcompany-exercise.core-test
  (:require [clojure.test :refer :all]
            [trysomecoolcompany-exercise.core :refer :all]))

(def db-conf {:host "localhost"
              :port 5432
              :dbname "postgres"
              :user "andrea"
              :password ""
              :table "test_person"})
```

```
(def url
  (-> "test.xml"
      clojure.java.io/resource
      str))

(deftest xml->clj-vector-maps-test
  (testing "xml->clj should produce a list of maps with keys :fname :lname :dob :phone"
    (let [expected (list {:fname "JOHN"
                          :lname "DOE"
                          :dob "2002-02-01"
                          :phone "9548938821"
                         }
                         {:fname "Fiona"
                          :lname "Lullaby"
                          :dob "2010-02-01"
                          :phone "9548938822"
                         }
                         {:fname "XX"
                          :lname "YY"
                          :dob "2000-02-01"
                          :phone "9548138821"
                         })]
    (is (= (xml->clj url) expected)))))

(deftest xml->clj-keys-test
  (testing "xml->clj->db should contain only :fname :lname :dob :phone"
    (is (= (set (mapcat keys (xml->clj url))) #{:fname :lname :dob :phone}))))

(deftest clj-add-person-db-test
  (testing "xml-add-person-db should add a new person in the database from a person map"
    (let [db (connect-db db-conf)
          t (db-conf :table)
          person {:fname "XX"
                  :lname "YY"
                  :dob "2000-02-01"
                  :phone "9548138821"
                  }
          expected {:updated 1 :rows []}]
      (is (= (clj-add-person-db db t person) expected))
```

```
      (clj-del-person-db db t person)))) ;cleaning db after usage
```

Now code to pass them:

```
(ns trysomecoolcompany-exercise.core
  (:require [clojure.zip :as zip]
            [clojure.data.zip.xml :refer :all]
            [clojure.data.xml :as data.xml]
            [clojure.core.async :as async :refer [<!!]]
            [postgres.async :refer :all]))

(defn get-zipper
  "creates a zipper from an xml url"
  [url]
  (-> url
      clojure.java.io/input-stream
      data.xml/parse
      zip/xml-zip))

(defn xml->clj
  "transforms zipper nodes in maps"
  [url]
  (let [zxml (get-zipper url)
        members (xml-> zxml :members :member)]
    (for [member members  :let [fname (xml1-> member :first-name text)
                                lname (xml1-> member :last-name text)
                                dob (xml1-> member :date-of-birth text)
                                phone (xml1-> member :phone text)]
          ]
  {:fname fname :lname lname :dob dob :phone phone})))

(defn connect-db
  "creates a db connection"
  [dbc]
  (open-db {:hostname (dbc :host)
                  :port (dbc :port)
                  :database (dbc :dbname)
                  :username (dbc :user)
                  :password (dbc :password)
```

```
                  :pool-size 25}))

(defn close-db
  "closes a db connection"
  [db]
  (close-db! db))

(defn clj-add-person-db
  "add or update person to database's table"
  [db t p]
  (<!! (insert! db {:table t} p)))

(defn clj-del-person-db
  "delete person from database's table"
  [db t p]
  (<!! (execute! db [(str "delete from " t " where fname='"(p :fname)"' and lname='"(p
```

6.4 DONE feature 2: do not create a person record if exists in the db

- CLOSING NOTE *[2017-01-19 Thu 12:34]*
  woah, this was a little of postgres study: I like upsert (on conflict do set...)!

here the best choice is to add a primary key on name lastname and dob. However, we probably want to use an upsert, because using two postgres commands allows race conditions: `http://stackoverflow.com/questions/1109061/insert-on-duplicate-update-in-postgresql/8702291#8702291`

Let's add a test case for upsert:

```
(ns trysomecoolcompany-exercise.core-test
  (:require [clojure.test :refer :all]
            [trysomecoolcompany-exercise.core :refer :all]))

(def db-conf {:host "localhost"
              :port 5432
              :dbname "postgres"
              :user "andrea"
              :password ""
              :table "test_person"})
```

```
(def url
  (-> "test.xml"
      clojure.java.io/resource
      str))

(deftest xml->clj-vector-maps-test
  (testing "xml->clj should produce a list of maps with keys :fname :lname :dob :phone"
    (let [expected (list {:fname "JOHN"
                          :lname "DOE"
                          :dob "2002-02-01"
                          :phone "9548938821"
                         }
                         {:fname "Fiona"
                          :lname "Lullaby"
                          :dob "2010-02-01"
                          :phone "9548938822"
                         }
                         {:fname "XX"
                          :lname "YY"
                          :dob "2000-02-01"
                          :phone "9548138821"
                         })]
    (is (= (xml->clj url) expected)))))

(deftest xml->clj-keys-test
  (testing "xml->clj->db should contain only :fname :lname :dob :phone"
    (is (= (set (mapcat keys (xml->clj url))) #{:fname :lname :dob :phone}))))

(deftest clj-add-person-db-test
  (testing "clj-add-person-db should add a new person in the database from a person map"
    (let [db (connect-db db-conf)
          t (db-conf :table)
          person {:fname "XX"
                  :lname "YY"
                  :dob "2000-02-01"
                  :phone "9548138821"
                  }
          expected {:updated 1 :ignored 0}]
      (clj-del-person-db db t person) ; cleaning db before usage
      (is (= (clj-add-person-db db t person) expected))
```

```
      (clj-del-person-db db t person)))) ;cleaning db after usage

(deftest clj-add-same-person-twice-db-test
  (testing "clj-add-people-db should add a new person in the database from a person map
    (let [db (connect-db db-conf)
          t (db-conf :table)
          person {:fname "X"
                  :lname "Y"
                  :dob "1999-11-01"
                  :phone "9128138821"
                  }
          expected {:updated 1 :ignored 1}]
          (clj-del-person-db db t person) ; cleaning db before usage
          (is (= (clj-add-people-db db t person person) expected))
          (clj-del-person-db db t person)))) ;cleaning db after usage

(deftest clj-add-two-people-db-test
  (testing "clj-add-people-db should add two new person in the database from a person m
    (let [db (connect-db db-conf)
          t (db-conf :table)
          p1 {:fname "ZX"
                  :lname "Y"
                  :dob "1899-11-01"
                  :phone "9128138821"
                  }
          p2 {:fname "FX"
                  :lname "Y"
                  :dob "1129-11-01"
                  :phone "9128138821"
                  }
          expected {:updated 2 :ignored 0}]
          (clj-del-person-db db t p1) ; cleaning db before usage
          (clj-del-person-db db t p2)
          (is (= (clj-add-people-db db t p1 p2) expected))
          (clj-del-person-db db t p1) ; cleaning db after usage
          (clj-del-person-db db t p2))))
```

Now code to pass the test:

```
(ns trysomecoolcompany-exercise.core
```

```
  (:require [clojure.zip :as zip]
            [clojure.data.zip.xml :refer :all]
            [clojure.data.xml :as data.xml]
            [clojure.core.async :as async :refer [<!!]]
            [postgres.async :refer :all]))

(defn get-zipper
  "creates a zipper from an xml url"
  [url]
  (-> url
      clojure.java.io/input-stream
      data.xml/parse
      zip/xml-zip))

(defn xml->clj
  "transforms zipper nodes in maps"
  [url]
  (let [zxml (get-zipper url)
        members (xml-> zxml :members :member)]
    (for [member members  :let [fname (xml1-> member :first-name text)
                                lname (xml1-> member :last-name text)
                                dob (xml1-> member :date-of-birth text)
                                phone (xml1-> member :phone text)]
          ]
  {:fname fname :lname lname :dob dob :phone phone})))

(defn connect-db
  "creates a db connection"
  [dbc]
  (open-db {:hostname (dbc :host)
                  :port (dbc :port)
                  :database (dbc :dbname)
                  :username (dbc :user)
                  :password (dbc :password)
                  :pool-size 25}))

(defn close-db
  "closes a db connection"
  [db]
  (close-db! db))
```

```
(defn clj-add-person-db
  "add or update person to database's table"
  [db t p]
  ; I needed to add a primary key over fname lname and dob to make
  ; this work
  (let [ks (keys p)
        kks (drop-last ks)
        pk (name (last ks))
        kcols (clojure.string/join " , " (map name kks))
        cols (clojure.string/join " , " (map name ks))
        vs (vals p)
        vals (str "'" (clojure.string/join "' , '" vs) "'") ;FIXME adding single quote
        pv (str "'" (last vs) "'")
        query (str
               "insert into " t
               " (" cols ") "
               "values (" vals ") "
               "on conflict (" kcols ") "
               "do update set " pk " = " pv "where "pv" <> " t "." pk ";")]
    (if (= {:updated 0 :rows []} (<!! (execute! db [query])))
      {:updated 0 :ignored 1}
      {:updated 1 :ignored 0})))


(defn clj-add-people-db
  "add or update person to database's table"
  [db t & ps]
  (let [rs (map #(clj-add-person-db db t %) ps)
        f (fn [acc r] {:updated (+ (r :updated) (acc :updated))
                       :ignored (+ (r :ignored) (acc :ignored))})]
    (reduce f {:updated 0 :ignored 0} rs)))

(defn clj-del-person-db
  "delete person from database's table"
  [db t p]
  (<!! (execute! db [(str "delete from " t " where fname='"(p :fname)"' and lname='"(p
```

6.5 DONE feature 3: update a person number if it exists in the db

- CLOSING NOTE *[2017-01-19 Thu 00:43]*
  done during feature 2

6.6 DONE feature 4: update all the people in the xml returning overall stats

- CLOSING NOTE *[2017-01-19 Thu 14:14]*

Let's add a test case for a function that transfers xml people to db people:

```
(ns trysomecoolcompany-exercise.core-test
  (:require [clojure.test :refer :all]
            [trysomecoolcompany-exercise.core :refer :all]))

(def db-conf {:host "localhost"
              :port 5432
              :dbname "postgres"
              :user "andrea" ;; FIXME change with a postgres user
              :password ""})

(def db (connect-db db-conf))

(def t "test_person")

(def url
  (-> "test.xml"
      clojure.java.io/resource
      str))
(def p123
  [{:fname "JOHN"
   :lname "DOE"
   :dob "2002-02-01"
   :phone "9548938821"
   }
  {:fname "Fiona"
   :lname "Lullaby"
   :dob "2010-02-01"
   :phone "9548938822"
```

```
   }
  {:fname "XX"
   :lname "YY"
   :dob "2000-02-01"
   :phone "9548138821"
   }])

(def p {:fname "XX"
        :lname "YY"
        :dob "2001-02-01"
        :phone "9548138821"
        })

(def person {:fname "X"
             :lname "Y"
             :dob "1919-11-01"
             :phone "9128138821"
             })

(def p1 {:fname "ZX"
         :lname "Y"
         :dob "1899-11-01"
         :phone "9128138821"
         })

(def p2 {:fname "FX"
         :lname "Y"
         :dob "1129-11-01"
         :phone "9128138821"
         })

(defn cool-down []
  (doall (map #(clj-del-person-db db t %) (conj p123 person p p1 p2)))
  (Thread/sleep 100)
  (close-db db))

; to clean database
(use-fixtures :once
  (fn [tests]
    (tests)
```

```
    (cool-down)))

(deftest xml->clj-vector-maps-test
  (testing "xml->clj should produce a list of maps with keys :fname :lname :dob :phone"
    (let [expected p123]
    (is (= (into [] (xml->clj url)) expected)))))

(deftest xml->clj-keys-test
  (testing "xml->clj->db should contain only :fname :lname :dob :phone"
    (is (= (set (mapcat keys (xml->clj url))) #{:fname :lname :dob :phone}))))

(deftest clj-add-person-db-test
  (testing "clj-add-person-db should add a new person in the database from a person map"
    (let [expected {:updated 1 :ignored 0}]
      (is (= (clj-add-person-db db t p) expected)))))

(deftest clj-add-same-person-twice-db-test
  (testing "clj-add-people-db should add a new person in the database from two same people"
    (let [expected {:updated 1 :ignored 1}]
          (is (= (clj-add-people-db db t [person person]) expected)))))

(deftest clj-add-two-people-db-test
  (testing "clj-add-people-db should add two new people in the database from two persons"
    (let [expected {:updated 2 :ignored 0}]
          (is (= (clj-add-people-db db t [p1 p2]) expected)))))

(deftest xml-db-transfer-people-test
  (testing "xml-db-transfer-people should insert not existing people in the db"
    (let [expected {:updated 3 :ignored 0}]
      (is (= (xml-db-transfer-people db t url) expected)))))
```

Now code to pass the test:

```
(ns trysomecoolcompany-exercise.core
  (:require [clojure.zip :as zip]
            [clojure.data.zip.xml :refer :all]
            [clojure.data.xml :refer [parse]]
            [clojure.data.xml :as data.xml]
            [clojure.core.async :as async :refer [<!!]]
            [postgres.async :refer :all]
```

```
            [clojure.string :refer [join]]))

(defn xml->clj
  "transforms zipper nodes in maps"
  [url]
  (let [get-members (fn [x] (->> x
                     clojure.java.io/input-stream
                     parse
                     :content
                     (filter #(= :member (:tag %)))))
        get-value (fn [key] (xml1-> (zip/xml-zip member) key text))]
    (for [member (get-members url) :let [fname (get-value :firstname)
                                         lname (get-value :lastname)
                                         dob   (get-value :date-of-birth)
                                         phone (get-value :phone)]
          ]
      {:fname fname :lname lname :dob dob :phone phone})))

(defn connect-db
  "creates a db connection"
  [dbc]
  (open-db {:hostname (:host dbc)
            :port (:port dbc)
            :database (:dbname dbc)
            :username (:user dbc)
            :password (:password dbc)
            :pool-size 25}))

(defn close-db
  "closes a db connection"
  [db]
  (close-db! db))

(defn clj-add-person-db
  "add or update person to database's table"
  [db t p]
  ; I needed to add a primary key over fname lname and dob to make
  ; this work
  (let [ks (keys p)
        phone-k (name (last ks))
```

```
        kcols (join " , " (map name (drop-last ks)))
        cols (join " , " (map name ks))
        vs (vals p)
        vals (str "'" (join "' , '" vs) "'") ;FIXME adding single quote works only if i
        phone-v (str "'" (last vs) "'")
        query (str
               "INSERT INTO " t
               " (" cols ") "
               "VALUES (" vals ") "
               "ON CONFLICT (" kcols ") "
               "DO UPDATE SET " phone-k " = " phone-v "WHERE "phone-v" <> " t "." phone
               r (<!! (execute! db  [query]))] ;FIXME we could extend the returning map
    (if (= {:updated 1 :rows []} r)
      {:updated 1 :ignored 0}
      {:updated 0 :ignored 1})))


(defn clj-add-people-db
  "add or update person to database's table"
  [db t ps]
  (let [f (fn [acc r] (let [r {:updated (+ (r :updated) (acc :updated))
                               :ignored (+ (r :ignored) (acc :ignored))}]
                        (print (str "\r" "Insertion stats: " r " of " (+ (r :ignored)
                        r))]
    (reduce f {:updated 0 :ignored 0} (map #(clj-add-person-db db t %) ps))))

(defn clj-del-person-db
  "delete person from database's table"
  [db t p]
  (<!! (execute! db [(str "DELETE FROM " t " WHERE fname='"(p :fname)"' AND lname='"(p


(defn xml-db-transfer-people
  "moves xml people records to database people rows"
  [db t url]
  (clj-add-people-db db t (xml->clj url)))

(defn -main
  "This converts the somecoolcompany huge xml in db data (it requires a postgres userna
  [user]
```

```
  (let [url "/tmp/somecoolcompany-backend-test/update-file.xml"
        dbc {:host "localhost"
             :port 5432
             :dbname "postgres"
             :user user
             :password ""}
        db (connect-db dbc)
        t "person"]
    (println (str "\n" "Finished:" (xml-db-transfer-people db t url)))
    (close-db db)))
```

6.7 DONE refactor code further ("Write clean code")

- CLOSING NOTE *[2017-01-22 Sun 19:55]*
  I did refactoring before

```
(ns trysomecoolcompany-exercise.core
  (:require [clojure.zip :as zip]
            [clojure.data.zip.xml :refer :all]
            [clojure.data.xml :refer [parse]]
            [clojure.data.xml :as data.xml]
            [clojure.core.async :as async :refer [<!!]]
            [postgres.async :refer :all]
            [clojure.string :refer [join]]))

(defn xml->clj
  "transforms zipper nodes in maps"
  [url]
  (let [get-members (fn [x] (->> x
                     clojure.java.io/input-stream
                     parse
                     :content
                     (filter #(= :member (:tag %)))))
        get-value (fn [m k] (xml1-> (zip/xml-zip m) k text))]
    (for [member (get-members url) :let [fname (get-value member :firstname)
                                         lname (get-value member :lastname)
                                         dob   (get-value member :date-of-birth)
                                         phone (get-value member :phone)]
          ]
      {:fname fname :lname lname :dob dob :phone phone})))
```

```
(defn connect-db
  "creates a db connection"
  [dbc]
  (open-db {:hostname (:host dbc)
            :port (:port dbc)
            :database (:dbname dbc)
            :username (:user dbc)
            :password (:password dbc)
            :pool-size 25}))

(defn close-db
  "closes a db connection"
  [db]
  (close-db! db))

(defn clj-add-person-db
  "add or update person to database's table"
  [db t p]
  ; I needed to add a primary key over fname lname and dob to make
  ; this work
  (let [ks (keys p)
        phone-k (name (last ks))
        kcols (join " , " (map name (drop-last ks)))
        cols (join " , " (map name ks))
        vs (vals p)
        vals (str "'" (join "' , '" vs) "'") ;FIXME adding single quote works only if
        phone-v (str "'" (last vs) "'")
        query (str
               "INSERT INTO " t
               " (" cols ") "
               "VALUES (" vals ") "
               "ON CONFLICT (" kcols ") "
               "DO UPDATE SET " phone-k " = " phone-v "WHERE "phone-v" <> " t "." phone
               r (<!! (execute! db  [query]))] ;FIXME we could extend the returning ma
    (if (= {:updated 1 :rows []} r)
      {:updated 1 :ignored 0}
      {:updated 0 :ignored 1})))
```

```
(defn clj-add-people-db
  "add or update person to database's table"
  [db t ps]
  (let [f (fn [acc r] (let [r {:updated (+ (r :updated) (acc :updated))
                              :ignored (+ (r :ignored) (acc :ignored))}]
                        (print (str "\r" "Insertion stats: " r " of " (+ (r :ignored)
                        r))]
    (reduce f {:updated 0 :ignored 0} (map #(clj-add-person-db db t %) ps))))

(defn clj-del-person-db
  "delete person from database's table"
  [db t p]
  (<!! (execute! db [(str "DELETE FROM " t " WHERE fname='"(p :fname)"' AND lname='"(p


(defn xml-db-transfer-people
  "moves xml people records to database people rows"
  [db t url]
  (clj-add-people-db db t (xml->clj url)))

(defn -main
  "This converts the somecoolcompany huge xml in db data (it requires a postgres usern
  [user]
  (let [url "/tmp/somecoolcompany-backend-test/update-file.xml"
        dbc {:host "localhost"
             :port 5432
             :dbname "postgres"
             :user user
             :password ""}
        db (connect-db dbc)
        t "person"]
    (println (str "\n" "Finished:" (xml-db-transfer-people db t url)))
    (close-db db)))
```

6.8 DONE "minimize the overall run time of the merge process"

- CLOSING NOTE *[2017-01-19 Thu 17:49]*
  maybe I can increase the pool of connections? 25 does 1000 insertion every 5 seconds cc

- Note taken on *[2017-01-18 Wed 11:53]*
  I think core.async should do a good job using threads. Maybe I can create more threads somehow? Check postgres.async doc.

`http://blog.korny.info/2014/03/08/xml-for-fun-and-profit.html`

With a pool of 100 inserting the xml data in an empty db (in memory) runs in:

```
real 11m1.304s
user 7m28.874s
sys 0m52.740s
```

with a pool of 1500:

```
real 10m52.449s
user 7m27.015s
sys 0m52.778s
```

so it is irrelevant.

By the way tested on the real database:

```
Insertion stats: {:updated 927147, :ignored 572853} of 1500000
Statistics:{:updated 927147, :ignored 572853}


real 9m4.016s
user 6m28.890s
sys 0m44.068s
```

6.9 DONE check performance (" Reason about performance and memory usage")

- CLOSING NOTE *[2017-01-22 Sun 20:25]*
  constant usage of cpu

  and time

  Insertion stats: {:updated 927147, :ignored 572853} of 1500000 Statistics:{:updated 927147, :ignored 572853}

  real 9m4.016s user 6m28.890s sys 0m44.068s

Let's install sysstat

```
nix-env -iA nixos.sysstat
```

Lets get a performance stat of our process while it is running:

```
pidstat -p <pid-of-process-here> 2
```

Table 1:

time	UID	PID	%usr	%system	%guest	%CPU	CPU	Command
20:00:27	1000	15023	70.5	9.5	0.0	80.0	2	java
20:00:29	1000	15023	73.5	9.0	0.0	82.5	2	java
20:00:31	1000	15023	72.5	7.0	0.0	79.5	2	java
20:00:33	1000	15023	71.0	9.5	0.0	80.5	2	java
20:00:35	1000	15023	72.5	7.5	0.0	80.0	2	java
20:00:37	1000	15023	71.0	8.5	0.0	79.5	2	java
20:00:39	1000	15023	71.0	9.0	0.0	80.0	2	java
20:00:41	1000	15023	70.5	8.0	0.0	78.5	2	java
20:00:43	1000	15023	68.0	8.0	0.0	76.0	2	java
20:00:45	1000	15023	69.0	8.5	0.0	77.5	2	java
20:00:47	1000	15023	71.0	7.0	0.0	78.0	2	java
20:00:49	1000	15023	66.0	8.0	0.0	74.0	2	java
20:00:51	1000	15023	68.0	8.5	0.0	76.5	2	java
20:00:53	1000	15023	69.5	7.5	0.0	77.0	2	java
20:00:55	1000	15023	62.0	8.0	0.0	70.0	2	java
20:00:57	1000	15023	65.0	7.5	0.0	72.5	2	java
20:00:59	1000	15023	66.5	8.0	0.0	74.5	2	java
20:01:01	1000	15023	68.0	8.0	0.0	76.0	2	java
20:01:03	1000	15023	67.5	8.5	0.0	76.0	2	java
20:01:05	1000	15023	67.0	9.0	0.0	76.0	2	java
20:01:07	1000	15023	68.0	8.0	0.0	76.0	2	java
20:01:09	1000	15023	66.5	7.5	0.0	74.0	2	java
20:01:11	1000	15023	70.0	7.5	0.0	77.5	2	java
20:01:13	1000	15023	69.0	9.0	0.0	78.0	2	java
20:01:15	1000	15023	70.5	6.5	0.0	77.0	2	java
20:01:17	1000	15023	70.0	8.0	0.0	78.0	2	java
20:01:19	1000	15023	62.0	8.5	0.0	70.5	2	java
20:01:21	1000	15023	66.0	7.0	0.0	73.0	2	java
20:01:23	1000	15023	68.0	8.0	0.0	76.0	2	java
20:01:25	1000	15023	68.5	8.0	0.0	76.5	2	java
20:01:27	1000	15023	68.5	8.0	0.0	76.5	2	java
20:01:29	1000	15023	69.0	7.5	0.0	76.5	2	java
20:01:31	1000	15023	68.5	8.0	0.0	76.5	2	java
20:01:33	1000	15023	67.5	7.5	0.0	75.0	2	java
20:01:35	1000	15023	68.0	8.0	0.0	76.0	2	java
20:01:37	1000	15023	69.0	7.5	0.0	76.5	2	java
20:01:39	1000	15023	63.5	8.0	0.0	71.5	2	java
20:01:41	1000	15023	69.0	8.0	0.0	77.0	2	java
20:01:43	1000	15023	67.0	8.0	0.0	75.0	2	java
20:01:45	1000	15023	67.5	31 9.0	0.0	76.5	2	java
20:01:47	1000	15023	68.5	6.5	0.0	75.0	2	java
20:01:49	1000	15023	67.0	9.0	0.0	76.0	2	java
20:01:51	1000	15023	68.5	8.0	0.0	76.5	2	java
20:01:53	1000	15023	69.0	6.5	0.0	75.5	2	java
20:01:55	1000	15023	69.0	8.0	0.0	77.0	2	java
20:01:57	1000	15023	69.0	7.5	0.0	76.5	2	java
20:01:59	1000	15023	69.5	6.5	0.0	76.0	2	java

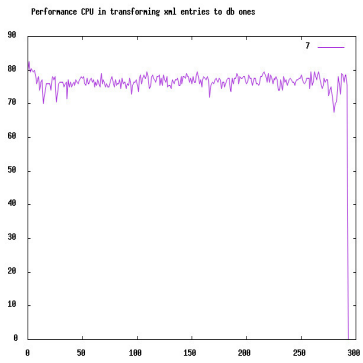

Figure 1: Performance in inserting xml entries in db