# UR2SQL.lhs

Jost Berthold

July 8, 2009

To compile this file: `$ghc --make -O2 UR2SQL.lhs` LaTeXcan be used to produce documentation: use the fancyvrb package, define a verbatim environment named "code", and a command `\cd` (similar to `\texttt`). and add some frame around this file (begin document and such). You can use `\input`.

Read usage records, convert data if necessary, insert into a database.

This file was written as a top-down design prototype, should phps. be split into several modules later. Some functionality,however, needs to be in tight sync: Parsing functions and SQL database schema are connected.

## Importing libraries, skip for "the big picture"

This imports list should be reduced to what is actually needed later (ghc has an option `-ddump-minimal-imports` to help).

We are using the HDBC library for database access from Haskell, and its ODBC driver, as well as the light-weight XML library XML.Light. Both available from `hackage.haskell.org`[1].

*compiled code*

```
{-# OPTIONS -cpp #-}
module Main where

-- dependencies to haskell libraries (not included in GHC)
import Database.HDBC       -- interface
import Database.HDBC.ODBC  -- driver
import Text.XML.Light      -- tiny XML tool set

-- Haskell libraries included in GHC:
import System
import Data.Time     -- requires extra instance SqlData for UTCTime

import System.Time   -- old stuff, to be replaced by Data.Time

import System.Locale -- locale for Time libraries
import Data.Maybe
import Data.Char
import Control.Monad.State
import qualified Control.Exception as E

-- debugging
#ifdef TRACE
import Debug.Trace -- logs to stderr when evaluating expressions...
#else
trace _ x = x
#endif
```

[1] The current version of this code uses library versions suitable for GHC-6.8.3, not the latest GHC-6.11 (6.12 when released). APIs should not change too much, but be aware.

### This is what we want to do:

*compiled code*

```
usage :: IO()
usage = getProgName >>= \n ->
        putStrLn ("Usage: #>" ++ n ++ " <usage record file>")

main = do args <- getArgs
          if null args
           then usage
           else do urFile <- readFile (head args)
                   let usageRecords = parseXMLDoc urFile
                       decodedURs   = case usageRecords of
                         Nothing -> error ("Parsing " ++ head args ++ " failed. "
                                           ++ "Not a usage record file?")
                         Just urs -> map decodeUR (splitURs urs)
                   db <- connectODBC databaseODBC
                   insertAllInto db decodedURs
                   disconnect db
#ifdef DEBUG
      where connectODBC dbstring = do putStrLn ("### connect "
                                                ++ dbstring ++ " ###")
                                      return (take 20 dbstring)
            disconnect db = putStrLn ("### disconnect " ++ db ++ " ###")
#endif
```

### Usage records as a Haskell data structure

This data structure needs to be filled. Here is the point where XML unmarshaling and database schema need to fit together. Thus, we should not modularise this part of the code, or at least keep this section together in one module.

Every optional value is simply a `Maybe` type, translated into NULL by the SQL interface later.

*compiled code*

```
data UsageRecord = UsageRecord { urId         :: String  -- mandatory
                               , urCreateTime :: Maybe UTCTime
                               -- mandatory for us, but defaulted
                               , urStatus     :: String -- mandatory
                               , urUser       :: User -- mandatory for us
                                                 -- (GlobalUserName)
                               , urJob        :: Job
                               , urStartTime  :: Maybe UTCTime
                               , urEndTime    :: Maybe UTCTime
                               , urWallDuration :: Maybe Double
                               , urCharge     :: Maybe Double
                               , urNodeCount  :: Maybe Int
                               -- and many more other fields...
                               , urXml        :: String -- verbatim XML
                               }
                 deriving (Read, Eq, Show)

data User = User { uGlobal ::String, uLocal ::Maybe String }
              -- global user name mandatory
                 deriving (Read, Eq, Show)

data Job  = Job  { jGlobal, jLocal ::Maybe String }
                 deriving (Read, Eq, Show)
```

The Haskell type should map to the database schema (the respective table for Usage Records). It is, however, possible to

- have additional unused (NULL) fields in the database table
- have fields in the usage record structure which are not intended to reach the database

The connecting point is the insert statement (values used) and the `extractFields` function, which is expected to produce a list of matching length for the prepared statement to insert the values.

*compiled code*

```
-- fields count and types should fit the insert statement, which
-- should in turn fit the DB schema.
-- if we could reify the data types inside UsageRecord, this could be
-- automatic. For now, it has to be modified manually when necessary.
fields :: [(String,String)] -- list of field names and types
fields = [ ("RecordId"      ,"VARCHAR(512) NOT NULL")
         , ("CreateTime"    ,"TIMESTAMP DEFAULT CURRENT_TIMESTAMP")
                         -- defaults to insertion time if null
         , ("GlobalUserName","VARCHAR(512) NOT NULL")
         , ("LocalUserId"   ,"VARCHAR(512)")
         , ("GlobalJobId"   ,"VARCHAR(512)")
         , ("LocalJobId"    ,"VARCHAR(512)")
         , ("Status"        ,"VARCHAR(32) NOT NULL")
         , ("StartTime"     ,"DATETIME") -- optional
         , ("EndTime"       ,"DATETIME") -- optional
         , ("WallDuration"  ,"DOUBLE") -- parsed from a duration
         , ("Charge"        ,"DOUBLE")
         , ("NodeCount"     ,"SMALLINT")
         -- many more other fields...
         , ("XML"           ,"TEXT NOT NULL") -- length 2^16, should do?
         ]

-- list of SqlValues. Field types are checked implicitly by the
-- respective UsageRecord field types.
extractFields :: UsageRecord -> [SqlValue]
extractFields ur =    [ toSql (urId ur)
                      , toSql (urCreateTime ur)
                      , toSql (uGlobal (urUser ur))]
                   ++ map toSql -- all "maybe String"
                        [ uLocal (urUser ur)
                        , jGlobal (urJob ur)
                        , jLocal (urJob ur)]
                   ++ [ toSql (urStatus ur)
                      , toSql (urStartTime ur)
                      , toSql (urEndTime ur)
                      , toSql (urWallDuration ur) -- seconds, as double
                      , toSql (urCharge ur)
                      , toSql (urNodeCount ur)
                      -- many more other fields...
                      , toSql (urXml ur)
                      ]

-- the matching statement:
insertPrepStatement :: String
insertPrepStatement = "insert into " ++ table
                      ++ "\n\t(" ++ sepBy ", " (map fst fields)
                      ++ ") values ( "
                      ++ sepBy ", " (replicate (length fields) " ? ")
                      ++ ");\n"

-- for building the statement:
table :: String
table = "JobUsageRecord"

-- a helper function: yield a single string where the list elements
```

```
-- are separated by the string given as the first argument.
sepBy :: String -> [String] -> String
sepBy sep strs | null strs = ""
               | otherwise = foldl1 (\a b -> a ++ sep ++ b) strs
--sepBy _ [] = ""
--sepBy _ [x] = x
--sepBy c (x:rest) = x ++ c : sepBy c rest
```

## Data Definition Statements

As we need to fix the table format anyway, we can also yield the matching table definition when desirable.

*compiled code*

```
-- statement to initialise the table:
initTableStatement :: String
initTableStatement = "CREATE TABLE " ++ table ++ "\n\t("
                     ++ sepBy "\n\t, "
                            ([ name ++ " " ++ typ
                               | (name,typ) <- fields ]
                             ++ indexOpts)
                     ++ "\n\t);"

-- index and key options for table creation
indexOpts = [ "PRIMARY KEY (RecordId)"
            , "INDEX time(CreateTime)"
            , "INDEX user (GlobalUserName)"
            , "INDEX exectime(StartTime,EndTime)"
            ]
-- could add the constraints here... better in the fields definition.
```

It would be nice to record versioned changes to the schema by concatenating DDL statements which capture the changes. However, this requires to select the matching version and only execute the new changes. And all would become unreadable. As we have just one live system...For now, manual work!

## Decoding Usage Record XML

Parsing XML data into the Usage record data structure is a separable task. Important, however: the Maybe type for missing fields.

We usually decode the data inside `JobUsageRecord`, but can first break up a single list node `UsageRecords` into its child elements.

*compiled code*

```
splitURs :: Element -> [Element]
splitURs e | qName(elName e) == "UsageRecords"
                        = filter isUR (elChildren e)
           | otherwise = error "unexpected XML content (no UsageRecords)"
    where isUR :: Element -> Bool
          isUR e = qName (elName e) == "JobUsageRecord"

decodeUR :: Element -> UsageRecord
decodeUR uRecord
    | qName (elName uRecord) /= "JobUsageRecord"
                = error ("unexpected content: " ++ show uRecord)
    | noRecId   = error "no recordID found"
    | otherwise = UsageRecord { urId = fromJust mId
                              , urCreateTime = mTime
```

```
                          , urStatus = status, urUser = user
                          , urJob = job
                          , urStartTime = start, urEndTime = end
                          , urWallDuration = time
                          , urCharge = charge, urNodeCount = nodes
                          , urXml = ppElement uRecord
                                    -- pretty-printed XML
                          }
  where mRecIdElem = findChildWith "RecordIdentity" uRecord
        recIdElem = fromJust mRecIdElem
        idNamesp = elName recIdElem
        mId   = findAttr (idNamesp{qName="recordId"}  ) recIdElem
        noRecId = case (mRecIdElem, mId) of
                    (Nothing,_) -> True
                    (_,Nothing) -> True
                    other       -> False
        mTime = do s <- findAttr (idNamesp{qName="createTime"}) recIdElem
                   parseT s
        status = stringInside "Status" uRecord
        userNode = findChildWith "UserIdentity" uRecord
        user = case userNode of
                 Nothing  -> User "unknown" Nothing
                 Just u ->
                   (User {uGlobal= stringInside "GlobalUserName" u
                         ,uLocal = maybeStringInside "LocalUserId" u })
        jobNode  = findChildWith "JobIdentity" uRecord
        job = case jobNode of
                 Nothing  -> Job {jGlobal=Nothing,jLocal=Nothing}
                 Just that ->
                    (Job {jGlobal= maybeStringInside "GlobalJobId" that
                         ,jLocal = maybeStringInside "LocalJobId" that})
        start = do s <- maybeStringInside "StartTime" uRecord
                   parseT s -- Maybe monad
        end = do s <- maybeStringInside "EndTime" uRecord
                 parseT s -- Maybe monad
        tStr = maybeStringInside "WallDuration" uRecord
        time = case tStr of
                 Nothing -> Nothing
                 Just t  -> Just (
                              if isDigit (head t)
                              then  read t
                              else if 'P' == head t
                                   then duration2Double t
                                   else error ("not a duration: " ++ t))
        charge = case (maybeStringInside "Charge"    uRecord) of
                   Nothing -> Nothing
                   Just s  -> Just (read s)
        nodes = case (maybeStringInside "NodeCount" uRecord) of
                  Nothing -> Nothing
                  Just s  -> Just (read s)
        -- match child element name, ignoring namespace URI and prefix
        findChildWith :: String -> Element -> Maybe Element
        findChildWith s e = filterChildName ((s==) . qName) e
        stringInside :: String -> Element -> String
        stringInside s e = case findChildWith s e of
                             Nothing -> ""
                             Just n  -> strContent n
        maybeStringInside :: String -> Element -> Maybe String
        maybeStringInside s e = do x <- findChildWith s e
                                   return (strContent x)
```

The types we are using require some additional helper functions:

- Decoding an XML `TimeDuration` into fractional seconds:

*compiled code*

```
duration2Double :: String -> Double
duration2Double ('P':dString)
    = let dayExtractFs = map getPart "YMD"
          (ymdDbl,rest) = runState (sequence dayExtractFs) dString
          timeStr = if null rest || head rest /= 'T'
                       then []
                       else tail rest
          timeExtractFs = map getPart "HMS"
          (hmsDbl,r2) = runState (sequence timeExtractFs) timeStr
          [y,m,d] = ymdDbl
          [h,min] = init hmsDbl
          (sec,parts) = properFraction (hmsDbl!!2)
          monthyeardays = -- compute from y and m...  timeDuration is
                          -- not well specified, so we approximate a
                          -- month by 30 days :(
                          y * 365 + m * 30
      in foldl sumTimes monthyeardays
             (zip [d,h,min,fromIntegral sec] [1,24,60,60])
         + parts
sumTimes :: Double -> (Double,Double) -> Double
sumTimes y (n,f) = f*y+n


getPart :: Char -> State String Double
getPart c = do input <- get
               let (num,rest) = span digitOrDecimal input
                   digitOrDecimal c = elem c "0123456789."
               if null rest
                  then return 0 -- not found, empty input
                  else if head rest == c
                         then do put (tail rest)
                                 return (read num)
                         else do put input
                                 return 0
```

This is actually not good; we put some effort into encoding a simpler format into this format before, now we decode it. The `arc-ur-logger` does not use `TimeDuration`, but merely fractional seconds (which, however, does not conform to the usage record format).

- The `UTCTime` data type we use for the time stamp does not have a conversion to SqlData (the older deprecated Haskell-98 time library is allegedly supported, but CalendarTime is convertied into SqlEpochTime, which creates an SQL NULL value). We create an instance of our own.

*compiled code*

```
instance SqlType UTCTime
    -- in fact, the driver returns us a string!
    where fromSql (SqlString s) = case parseTime defaultTimeLocale
                                                 dbTimeFormat s of
                                    Nothing  -> defaultDate
                                    Just utc -> utc
          toSql utc = SqlString (formatTime defaultTimeLocale dbTimeFormat utc)
dbTimeFormat :: String
dbTimeFormat = "%Y-%m-%d %H:%M:%S"
-- -- we are losing the picoseconds (SQL unable to set them)!

defaultDate :: UTCTime -- 1970-01-01 00:00:01
defaultDate  = UTCTime (fromGregorian 1970 1 1) 1

-- other instance, like the older CalendarTime (does not work):
--   where fromSql (SqlEpochTime s) = readTime defaultTimeLocale "%s"
--                                    ( show s )
```

```
--         toSql utc = SqlEpochTime (
--                     read (formatTime defaultTimeLocale "%s" utc))

-- another helper: parser function for time in ISO format.
parseT :: String -> Maybe UTCTime
parseT t = parseTime defaultTimeLocale "%Y-%m-%dT%H:%M:%S%QZ" t
```

### Database access

The database business is simply using the helper functions we have put together above. Care must be taken to catch SQL errors and not abort the whole statement sequence on a single error.

Open issue: How to get reasonable error details from HDBC.

*compiled code*

```
insertAllInto :: IConnection conn => conn -> [UsageRecord] -> IO ()
insertAllInto db urs = do let urVars = map extractFields urs::[[SqlValue]]
                          prep <-  prepare db insertPrepStatement
                                      -- ? marks for variables
                          sequence_ (map (executeWithCatch prep) urVars)
                          commit db
#ifdef DEBUG
       where prepare db stmt = -- compose a function to print statment
                               return (\s -> (putStrLn stmt >>
                                              putStrLn (show s)))
             commit db = putStrLn ("### commit " ++ db ++ " ###")
#endif

executeWithCatch stmt vars = E.catch (execute stmt vars >> return ())
                                     (\e -> putStrLn ("SQL error: "
                                                      ++ show e))
#ifdef DEBUG
       where execute prep var = prep var -- print it out only
#endif
```

About the database connection proper: we are using `connectODBC :: String -> IO Connection` from the ODBC driver. Documentation sez:

> For information on the meaning of the passed string, please see:
>
> `http://msdn2.microsoft.com/en-us/library/ms715433(VS.85).aspx`

Thank you... this is the grammar for the connection string:

```
connection-string ::= empty-string[;]
                  | attribute[;]
                  | attribute; connection-string
empty-string ::=
attribute ::= attribute-keyword=attribute-value
          | DRIVER=[{]attribute-value[}]
attribute-keyword ::= DSN | UID | PWD
                  | driver-defined-attribute-keyword
attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier
```

And as we found out on a couple of howto pages online: The DSN is the data source name, so should be configured on the machine as connecting to the server and database. User and password (UID,PWD) are clear. The whole string needs a trailing semicolon!

*compiled code*

```
databaseODBC :: String
databaseODBC = sepBy ";" [ "DSN=nr4-usagerecords"
                         , "UID=sqluser", "PWD=sqluser"
                         , ";"] -- trailing semicolon needed :P
```

## Using the database for agglomeration

We are most likely going to implement these parts in python later. Just for fun, we can select some agglomerated data from the tables:

*compiled code*

```
userSelectStatement :: String
userSelectStatement = unlines
      ["select GlobalUserName, count(*), sum(Charge), "
      ,"       month(StartTime), year(StartTime)"
      ,"from " ++ table
      ,"    where GlobalUserName = ? and StartTime is not NULL"
      ,"group by month(StartTime), year(StartTime)"
      ]

statsUser :: IConnection conn => conn -> String -> IO ()
statsUser db username
    = E.catch getStats (\e -> putStrLn ("SQL error: " ++ show e))
      where display :: [SqlValue] -> IO ()
            display [_,jobCountS,chargeSumS,monthS,yearS]
                    = do putStrLn
                           (show ((fromSql monthS,fromSql yearS)::(Int,Int))
                            ++ ": Charge " ++ show (fromSql chargeSumS::Double)
                            ++ " in " ++ show (fromSql jobCountS::Int) ++ " jobs"
                           )
            getStats = do results <- quickQuery db
                                       userSelectStatement [toSql username]
                          putStrLn (username ++
                                    "\n" ++ replicate (length username) '-')
                          mapM_ display results
                          putStrLn "-----------------------------"
```

This code does a select of all data for one particular GlobalUserName, grouping usage data (job count and charge sum) by month, and year.

*compiled code*

```
stats :: IO ()
stats = do db <- connectODBC databaseODBC
           E.catch
            (do users <- quickQuery db
                           ("select distinct GlobalUserName from "
                            ++ table)
                           []
                mapM_ (statsUser db . fromSql . head) users
            ) (\e -> do putStrLn ("Sql error: " ++ show e))
           disconnect db
```

### Furthermore...

It would be nice to extend the schema to more than one single table. Several improvements come into mind...

- Separate user table, foreign key links to JobUsageRecords The table could be filled by a trigger:

```
CREATE TRIGGER newUser
BEFORE INSERT ON JobUsageRecord
FOR EACH ROW BEGIN
  IF NOT (NEW.GlobalUserName IN
           (SELECT  GridUser.GlobalUserName FROM GridUser))
    THEN INSERT INTO GridUser(GlobalUserName , LocalUserId)
                  values (NEW.GlobalUserName, NEW.LocalUserId)
  END IF
END
```

  Later on, we might store VO information as well, fill a table VO in quite the same way, and add a GridUser_VO connection table as well. Performance will of course drop then.

- We should keep pre-agglomerated data for standard queries around to avoid repeatedly traversing the whole set.

- In the same spirit, saving away historic data into another table might pay off, accelerating the insertion and selection. Agglomerating queries have to be aware and split in two: current and historic.

All fairly standard...