**Mini-Project in databases: insurance model**



By Pinhas FEDIDAT, ID 7665692 and Yehuda KLIRS, ID 026618686.

With Aryeh Wiesen, Jerusalem College of Technology, 2013

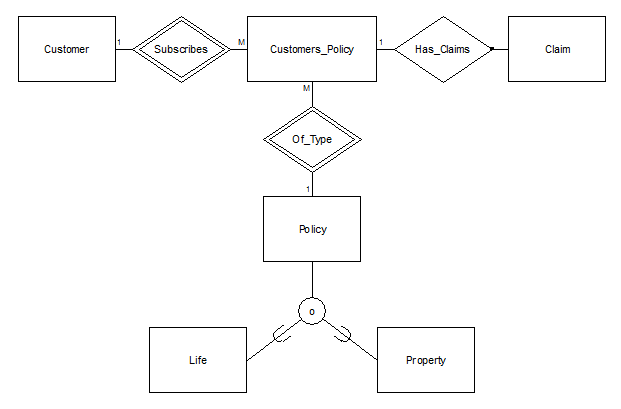
Stage 1 - Proposition of Organization – Insurance Model

The proposed subject for this mini-project is an insurance company model, organizing its customers, subscribed policies and claims. The company proposes insurance for various types of property (types of vehicles, houses or others) as well as life insurance depending on the occupation of the subscribing customer.

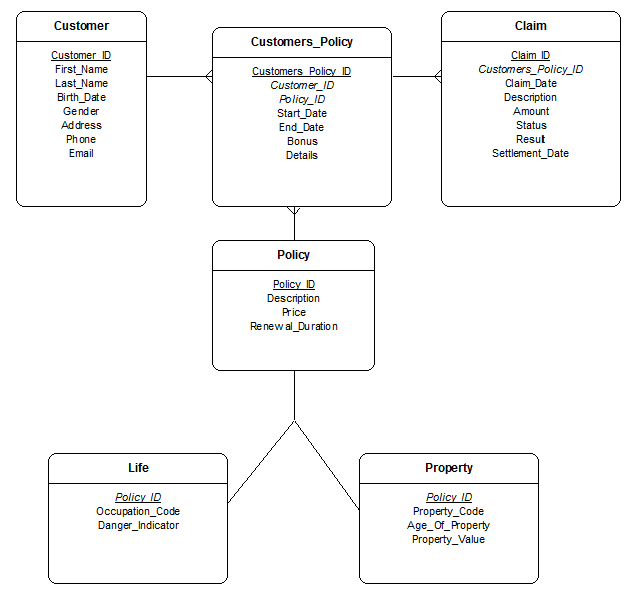
The database is composed of 6 tables::

* **Customers**: ID, first name, last name, birth date, gender and contact details (address, phone, email).
* **Policies**: Policy ID, description, price and duration of the policy until renewal (in months). There are 2 specifications of this table:
  + **Life insurance policies**: occupation code and respective danger indicator.
  + **Property insurance policies**: property code, age and value of property.
* **Subscribed policies (customers\_policies)**:customers\_policys\_ID, subscription start and end dates, bonus for long periods without claims and other details, customer and policy keys.
* **Complaints / claims**: claim ID, customers\_policys\_ID, .description, requested indemnification amount, status (e.g. pending, refused, accepted), and description of the result (e.g "Customer was fully indemnified"), claim and settlement dates.

**Entity–relationship model (ERD)**



**Data Structure Diagram (DSD)**



**SQL Script**

*--*

*-- Target: Oracle*

*-- Syntax: sqlplus user@tnsnames\_entry/password @filename.sql*

*--*

*-- Date : Apr 04 2013 15:07*

*-- Script Generated by Database Design Studio 2.21.3*

*--*

*--*

*-- Create Table : 'Customer'*

*-- Customer\_ID :*

*-- First\_Name :*

*-- Last\_Name :*

*-- Birth\_Date :*

*-- Gender :*

*-- Address :*

*-- Phone :*

*-- Email :*

*--*

CREATE TABLE Customer (

Customer\_ID NUMBER(38) NOT NULL,

First\_Name VARCHAR2(20) NOT NULL,

Last\_Name VARCHAR2(20) NOT NULL,

Birth\_Date DATE NOT NULL,

Gender NUMBER(1) NOT NULL,

Address VARCHAR2(100) NOT NULL,

Phone VARCHAR2(20) NOT NULL,

Email VARCHAR2(100) NOT NULL,

CONSTRAINT pk\_Customer PRIMARY KEY (Customer\_ID))

/

*--*

*-- Create Table : 'Policies'*

*-- Policy\_ID :*

*-- Description :*

*-- Price :*

*-- Renewal\_Duration :*

*--*

CREATE TABLE Policies (

Policy\_ID NUMBER(38) NOT NULL,

Description CLOB NOT NULL,

Price NUMBER(38) NOT NULL,

Renewal\_Duration NUMBER(38) NOT NULL,

CONSTRAINT pk\_Policies PRIMARY KEY (Policy\_ID))

/

*--*

*-- Create Table : 'Customers\_Policies'*

*-- Customers\_Policys\_ID :*

*-- Customer\_ID : (references Customer.Customer\_ID)*

*-- Policy\_ID : (references Policies.Policy\_ID)*

*-- Start\_Date :*

*-- End\_Date :*

*-- Bonus :*

*-- Details :*

*--*

CREATE TABLE Customers\_Policies (

Customers\_Policys\_ID NUMBER(38) NOT NULL,

Customer\_ID NUMBER(38) NOT NULL,

Policy\_ID NUMBER(38) NOT NULL,

Start\_Date DATE NOT NULL,

End\_Date DATE NOT NULL,

Bonus FLOAT NOT NULL,

Details CLOB,

CONSTRAINT pk\_Customers\_Policies PRIMARY KEY (Customers\_Policys\_ID),

CONSTRAINT fk\_Customers\_Policies FOREIGN KEY (Customer\_ID)

REFERENCES Customer (Customer\_ID)

ON DELETE CASCADE,

CONSTRAINT fk\_Customers\_Policies2 FOREIGN KEY (Policy\_ID)

REFERENCES Policies (Policy\_ID)

ON DELETE CASCADE)

/

*--*

*-- Create Table : 'Life'*

*-- Policy\_ID : (references Policies.Policy\_ID)*

*-- Occupation\_Code :*

*-- Danger\_Indicator :*

*--*

CREATE TABLE Life (

Policy\_ID NUMBER(38) NOT NULL,

Occupation\_Code NUMBER(38) NOT NULL,

Danger\_Indicator FLOAT NOT NULL,

CONSTRAINT pk\_Life PRIMARY KEY (Policy\_ID),

CONSTRAINT fk\_Life FOREIGN KEY (Policy\_ID)

REFERENCES Policies (Policy\_ID))

/

*--*

*-- Create Table : 'Property'*

*-- Policy\_ID : (references Policies.Policy\_ID)*

*-- Property\_Code :*

*-- Age\_Of\_Property :*

*-- Property\_Value :*

*--*

CREATE TABLE Property (

Policy\_ID NUMBER(38) NOT NULL,

Property\_Code NUMBER(1) NOT NULL,

Age\_Of\_Property NUMBER(38) NOT NULL,

Property\_Value NUMBER(38) NOT NULL,

CONSTRAINT pk\_Property PRIMARY KEY (Policy\_ID),

CONSTRAINT fk\_Property FOREIGN KEY (Policy\_ID)

REFERENCES Policies (Policy\_ID))

/

*--*

*-- Create Table : 'Claims'*

*-- Claim\_ID :*

*-- Customers\_Policys\_ID : (references Customers\_Policies.Customers\_Policys\_ID)*

*-- Claim\_Date :*

*-- Description :*

*-- Amount :*

*-- Status :*

*-- Result :*

*-- Settlement\_Date :*

*--*

CREATE TABLE Claims (

Claim\_ID NUMBER(38) NOT NULL,

Customers\_Policys\_ID NUMBER(38) NOT NULL,

Claim\_Date DATE NOT NULL,

Description CLOB NOT NULL,

Amount NUMBER(38) NOT NULL,

Status NUMBER(1),

Result CLOB,

Settlement\_Date DATE,

CONSTRAINT pk\_Claims PRIMARY KEY (Claim\_ID),

CONSTRAINT fk\_Claims FOREIGN KEY (Customers\_Policys\_ID)

REFERENCES Customers\_Policies (Customers\_Policys\_ID)

ON DELETE CASCADE)

/

*--*

*-- Permissions for: 'public'*

*--*

GRANT ALL ON Customer TO public

/

GRANT ALL ON Policies TO public

/

GRANT ALL ON Customers\_Policies TO public

/

GRANT ALL ON Life TO public

/

GRANT ALL ON Property TO public

/

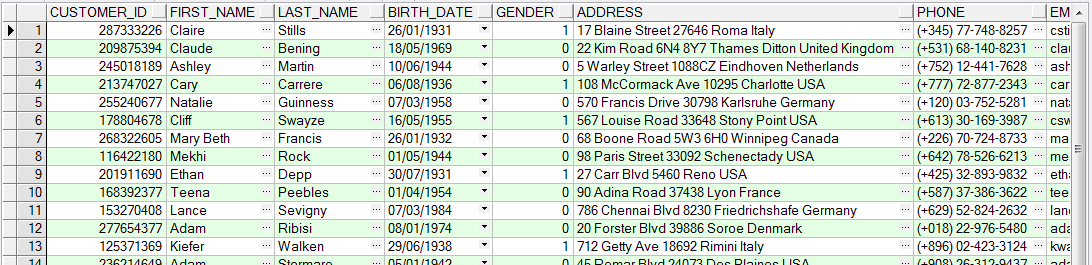
GRANT ALL ON Claims TO public

/

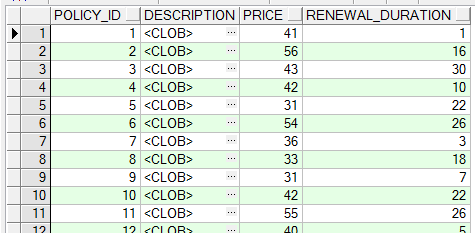
exit;

**Queries**

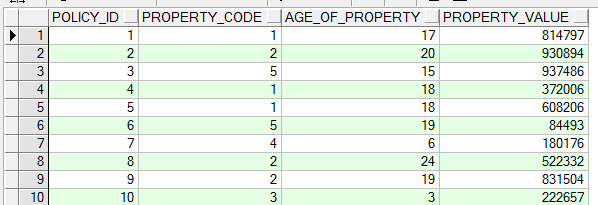
Customers



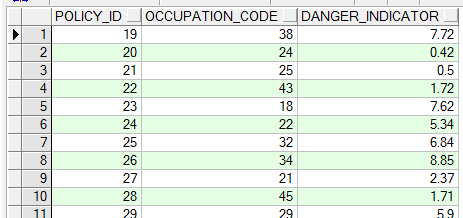
Policies



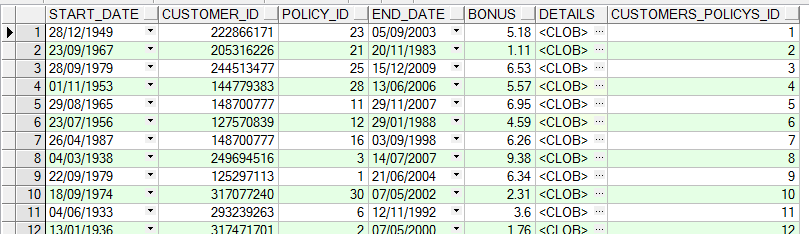
Property



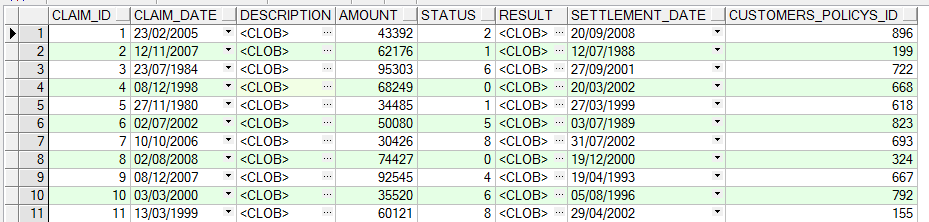
Life



Customers\_Policies



Claims



Level 2 – Queries, indexes, changes, constraints, views

**1. Eight queries that demonstrate practical use of the insurance database**

**1. Join, order by:** allows us to see all the policies subscribed by customers, together with customer information.

select customer\_id, policy\_id, start\_date

from customer natural join customers\_policy

order by start\_date asc;

**2. Join, aggregate, group by:** allows us to see the number of policies that customers subscribed to for each type of important policy (which have over 30 subscribers).

select policy.policy\_id, count(\*)

from policy left join customers\_policy

on policy.policy\_id = customers\_policy.policy\_id

group by policy.policy\_id

having count(\*) > 30;

**3. Sub-query, aggregate:** shows the sum of claims for policies subscribed after 01/01/2000.

select sum(amount) from (

select amount, status

from claim

where customers\_policy\_id in(

select customers\_policy\_id from customers\_policy

where start\_date > DATE '2000-01-01'

)

);

**4. Where, order by:** shows solved claims (where status is 0), sorted by submit date from oldest to most recent.

select \* from claim

where status = 0

order by claim\_date asc;

**5. Where, sub-query:** selects the name and emails of customers with the top ten bonuses.

select First\_Name, Last\_Name, email from Customer

where Customer\_ID in (

select Customer\_ID from (

select \* from Customers\_Policy

order by bonus desc)

where rownum <= 10);

**6. Group by, aggregate:** gives us a list of bad customers (of which the average of their bonuses is below 5).

select Customer\_ID, avg(bonus) from Customers\_Policy

group by Customer\_ID

having avg(bonus) < 5;

**7. Aggregate, group by:** shows the count of customers for each gender.

select gender, count(\*) from Customer

group by gender;

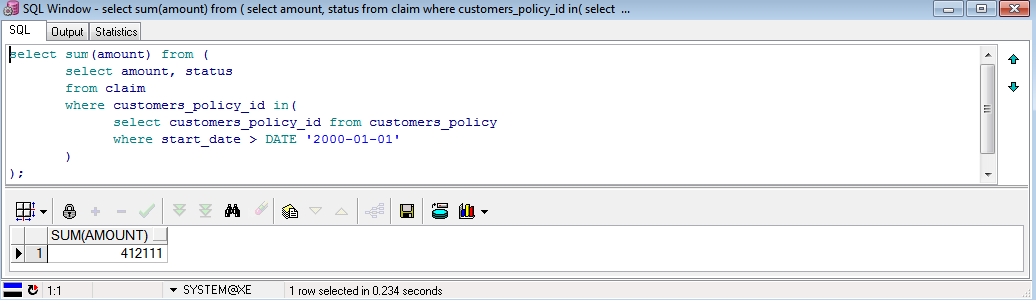
**8. Aggregate, group by:** shows the total price to charge for each customer.

select Customer\_ID, sum(Policy.price) from Customers\_Policy natural join Policy

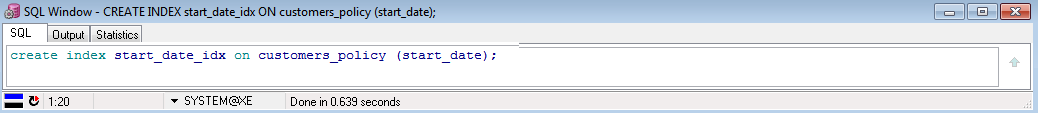
group by Customer\_ID;

**2. Index and speed improvement**

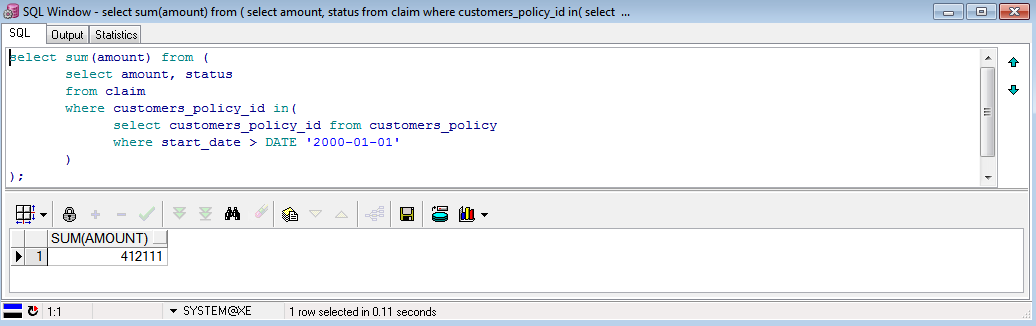
Here the query number 3 above is executed:



Then the Oracle server is stopped and started (cmd>net start/stop OracleServiceXE) in order to clear the buffers, which prevents unwanted cache optimization from the previous querying and we create an index over the customers\_policy.start\_date, which will optimize the where statement in the aforementioned query:



Finally, we run the query again. Notice the expected performance gain of 134ms (57%):



**3. Two Update and two Delete queries**

**1. Update** customer’s email address:

update customer

set email = 'dacruz@anomel.com'

where customer\_id = 237510106;

**2. Update** claim status:

update claim

set status = 0

where customers\_policy\_id = 153;

**3. Delete** claims related to customer’s policy 123:

delete claim

where customers\_policy\_id = 123;

**4. And then delete** the customer’s policy itself:

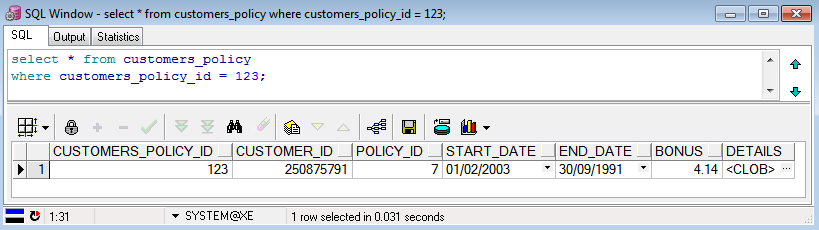
delete customers\_policy

where customers\_policy\_id = 123;

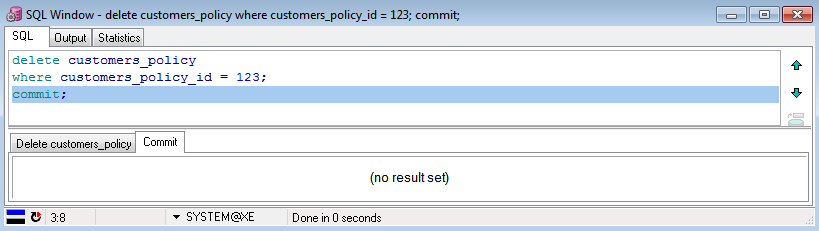
**4. Commit and rollback**

**1. Commit**

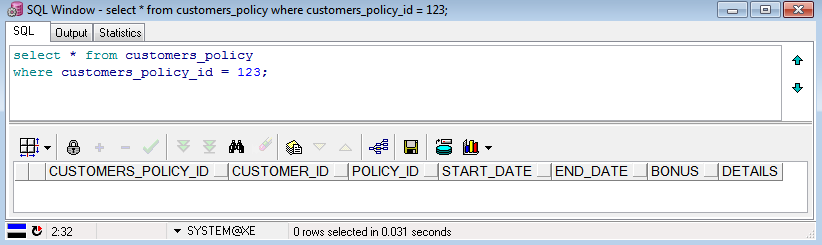
The customer’s policy record for ID 123 looks this way *before and after deletion*, as long as we don’t commit:



Then we execute the delete query and commit command:



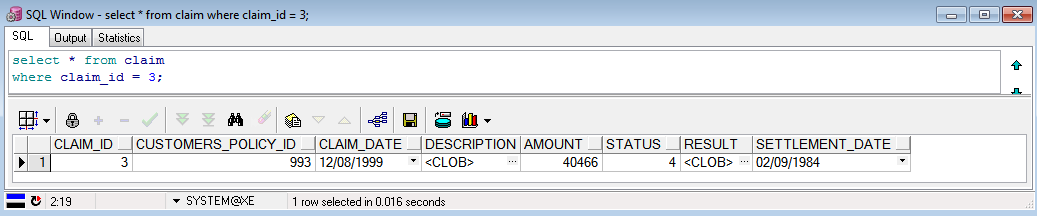
And this is what we get after commit:



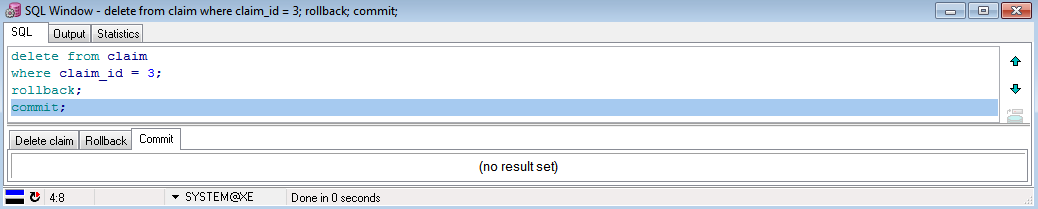
Of course, we can see that the customer’s policy record is now gone.

**2. Rollback**

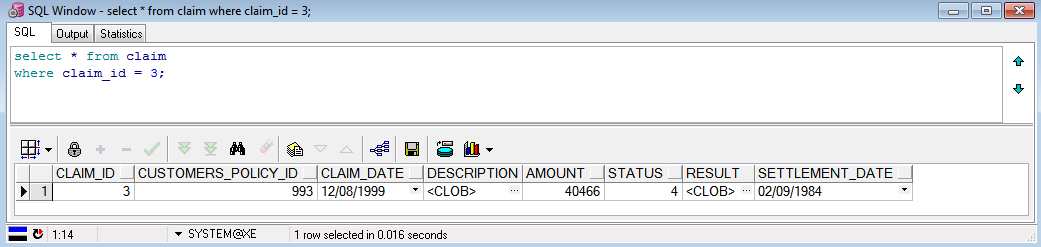
This is what we get before deletion:



Then we delete, rollback and commit (in order to show that even if we commit, as long as we rolled back, the changes before the rollback are cancelled):



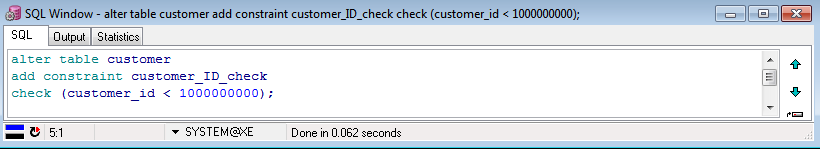
And finally query again:



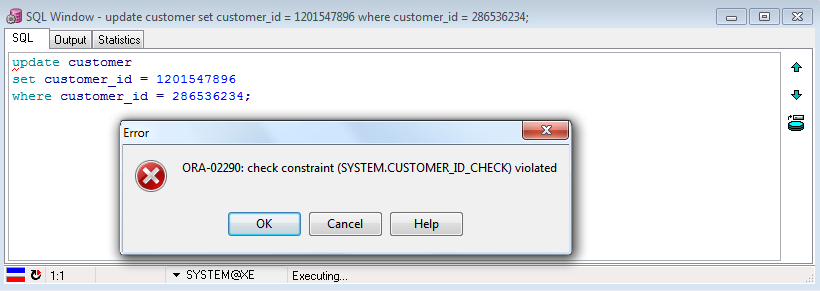
As we can see, the record has is still there despite the deletion, thanks to the rollback, just as expected.

**5. Constraints**

We define a new constraint, which checks that the customer\_ID has 9 digits from 0 to 9, or in other words, that the customer\_ID is strictly below 1000000000 (the lowest number with over 9 digits):

****

And now we update a customer’s ID with an illegal Id (1201547896, which has 10 digits):



And as expected, the update is refused, because the constraint customer\_ID\_check was violated.

**6. Views**

**1. Two views**

The first view, current\_customers\_policies, shows the ongoing policies of all customers:

create view current\_customers\_policies as

select \* from customers\_policy

where end\_date > sysdate;

The second view, accounts, shows the customer record information at a glance: ID, full name, number of subscribed policies and total price to pay for the month:

create view accounts as

select customer.customer\_id, customer.first\_name, customer.last\_name,

count(\*) as number\_of\_policies, sum(price) as total\_monthly\_price

from policy

join customers\_policy on customers\_policy.policy\_id = policy.policy\_id

join customer on customer.customer\_id = customers\_policy.customer\_id

group by customer.customer\_id, customer.first\_name, customer.last\_name;

**2. Two queries:**

For the first view, current\_customers\_policies, this query outputs the number of ongoing policies of a particular customer whose ID is 250875791:

select \* from current\_customers\_policies

where customer\_id = 250875791;

And for the second view, accounts, this query displays the number of customers, of customers’ policies, the total income and the average price for each customers’ policy:

select sum(customer\_id) as customers, sum(number\_of\_policies) as policies,

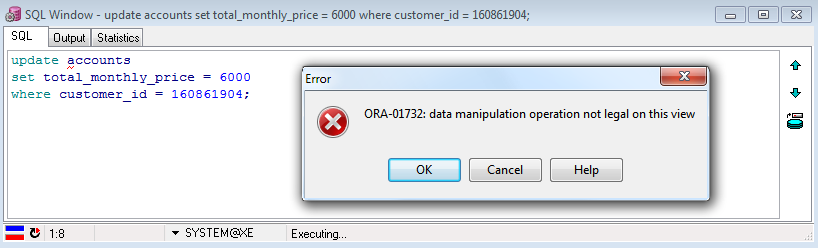
sum(total\_monthly\_price) as income,

sum(total\_monthly\_price)/sum(number\_of\_policies) as average\_policy\_cost

from accounts;

**3. View error**

We attempt to update the total price for a given customer in the accounts view.



Of course, this would break the data integrity since a view’s data is not stored but is just a saved query, and we cannot update the total directly but we have to update the price of the policy type itself.

Level 3 – PL (Programming Language) and Features

**1. Two procedures / functions**

**1. p\_updatePropertyPrice:** This procedure updates the price of any given property insurance policy type. It takes as parameters the base price, which is the minimum price for all policies, and the ID of the policy for which the price should be updated. It then calculates the final price for the property policy from the attributes in the property table (according to the type of property, its age, and its value) and updates it accordingly in the policy table.

create or replace procedure p\_updatePropertyPrice(i\_basePrice\_nr NUMBER, i\_policyID\_nr NUMBER) is

r\_prop property%ROWTYPE;

v\_propFactor\_nr NUMBER;

v\_ageOfProp\_nr NUMBER;

v\_propValue\_nr NUMBER;

v\_resultPrice\_nr NUMBER;

begin

*--Load record*

select \* into r\_prop from property where policy\_ID = i\_policyID\_nr;

*--Set type factor according to property code*

case r\_prop.property\_code

when 1 then

v\_propFactor\_nr := 1.5;

when 2 then

v\_propFactor\_nr := 2;

when 3 then

v\_propFactor\_nr := 2.5;

when 4 then

v\_propFactor\_nr := 3.5;

when 5 then

v\_propFactor\_nr := 5;

else

v\_propFactor\_nr := 1;

end case;

*--Set age of property*

v\_ageOfProp\_nr := r\_prop.age\_of\_property;

*--Set property value*

v\_propValue\_nr := r\_prop.property\_value;

*--Calculate resPrice*

v\_resultPrice\_nr := i\_basePrice\_nr + v\_propFactor\_nr\*v\_ageOfProp\_nr\*5 + v\_propValue\_nr/10000;\*

*--Update record*

update policy

set price = v\_resultPrice\_nr

where policy\_id = i\_policyID\_nr;

end p\_updatePropertyPrice;

**2. f\_sumForCustomer\_Nr:** This function returns the total price of all the policies of a particular customer ongoing at a given date. The parameters are the customer’s ID and the said date. A cursor then loops over the customer’s policies and sums the actual prices (policy type price multiplied by the customer’s bonus), and then returns the sum.

create or replace function f\_sumForCustomer\_Nr(i\_customerID\_nr NUMBER, v\_date\_dt DATE)

return NUMBER

is

cursor c\_custPolicy is

select \* from policy natural join customers\_policy

where customer\_ID = i\_customerID\_nr;

r\_custPolicy c\_custPolicy%ROWTYPE;

v\_sumPrices\_nr NUMBER := 0;

begin

*--Load cursor*

open c\_custPolicy;

loop

*--Load record*

fetch c\_custPolicy into r\_custPolicy;

exit when c\_custPolicy%NOTFOUND;

*--If policy date checks out*

if r\_custPolicy.start\_date < v\_date\_dt and r\_custPolicy.end\_date > v\_date\_dt then

*--Compute price of actual policy using price and bonus and add to sum*

v\_sumPrices\_nr := v\_sumPrices\_nr + r\_custPolicy.price \* (1 - 0.05 \* r\_custPolicy.bonus);

end if;

end loop;

close c\_custPolicy;

return(v\_sumPrices\_nr);

end f\_sumForCustomer\_Nr;

**2. Two Programs for the above procedures/functions**

**1.** This program updates the prices of all the property policies using the first procedure above (1.1). In order to do this, it uses a cursor that goes through all the rows from the property table and calls the procedure to update the relevant row in the policy table.

declare

cursor c\_prop is

select policy\_id from property;

r\_prop c\_prop%ROWTYPE;

begin

*--Load cursor*

open c\_prop;

loop

*--Load record*

fetch c\_prop into r\_prop;

exit when c\_prop%NOTFOUND;

*--Call p\_updatepropertyprice with base price 1000 and policy ID or each record*

p\_updatepropertyprice(i\_baseprice\_nr => 1000,

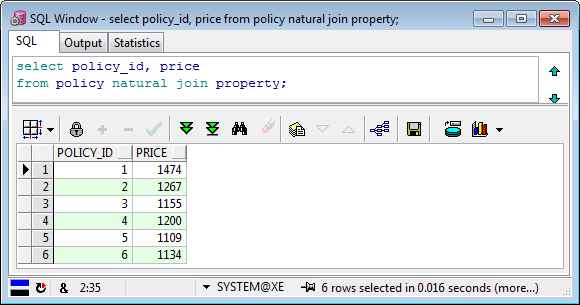
i\_policyid\_nr => r\_prop.policy\_id);

end loop;

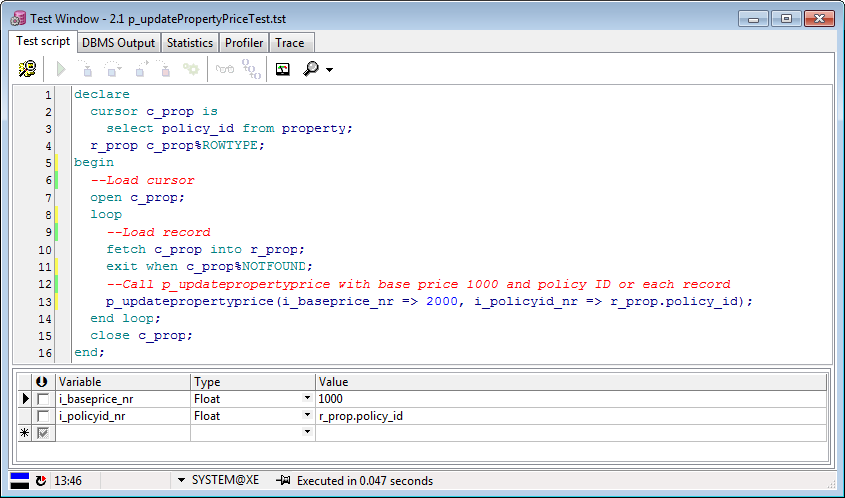
close c\_prop;

end;

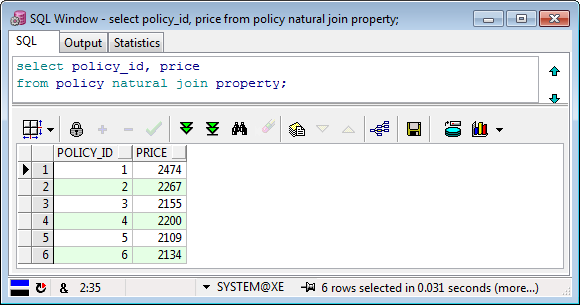
Let us check: first we query the database before the changes:



Then we run the procedure and commit:



And finally we query again with the same query and notice the updated results:



All as expected.

**2.** This program outputs the total price to pay for the customer\_id 109418126 as of 01/05/2006.

declare

*--Pick wanted customer ID*

v\_firstCustID\_Nr NUMBER := 109418126;

v\_result\_Nr NUMBER;

begin

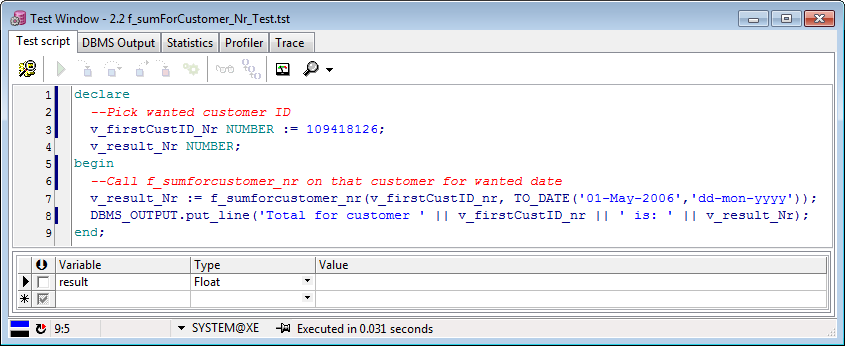
*--Call f\_sumforcustomer\_nr on that customer for wanted date*

v\_result\_Nr := f\_sumforcustomer\_nr(v\_firstCustID\_nr, TO\_DATE('01-May-2006','dd-mon-yyyy'));

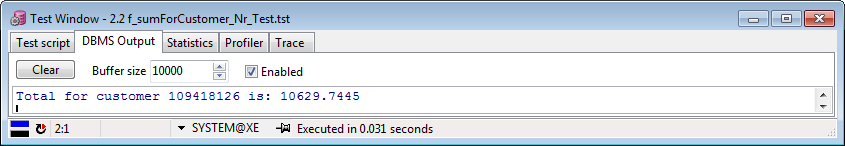
DBMS\_OUTPUT.put\_line('Total for customer ' || v\_firstCustID\_nr || ' is: ' || v\_result\_Nr);

end;

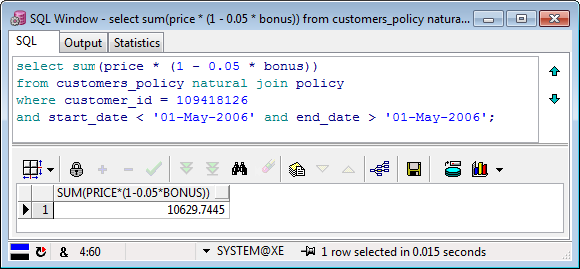
Let us check: we run the function:



Then we read the DBMS output tab:



And finally we can query the database to verify that the result checks out:



All good.

**3. Two triggers**

**1. pol\_checkDate** acts like a constraint which, instead of each insert into customers\_policy, checks whether the start\_date is more than a month later than today, in which case it calls a raise\_application\_error with an informative error message (unlike usual constraints) and rejects the insert query:

:

create or replace trigger pol\_checkDate

before insert or update on customers\_policy

for each row

declare

v\_error\_tx VARCHAR2(2000);

begin

*--If start date of policy more than a month from now*

if :new.start\_date > sysdate + 30 then

v\_error\_tx := 'policy start date for policy ' || :new.customers\_policy\_id || ' cannot be later than a month from today!';

*--Reject query and print error message*

raise\_application\_error(-20999, v\_error\_tx);

end if;

end property\_setPrice;

And here is a program that checks the constraints by entering one legal statement and an illegal one (as of this year, 2013):

declare

begin

*--Update policy with start\_date a while ago*

update customers\_policy

set start\_date = TO\_DATE('01-May-2006','dd-mon-yyyy')

where rownum = 1;

*--Update policy with start\_date more than a month from now*

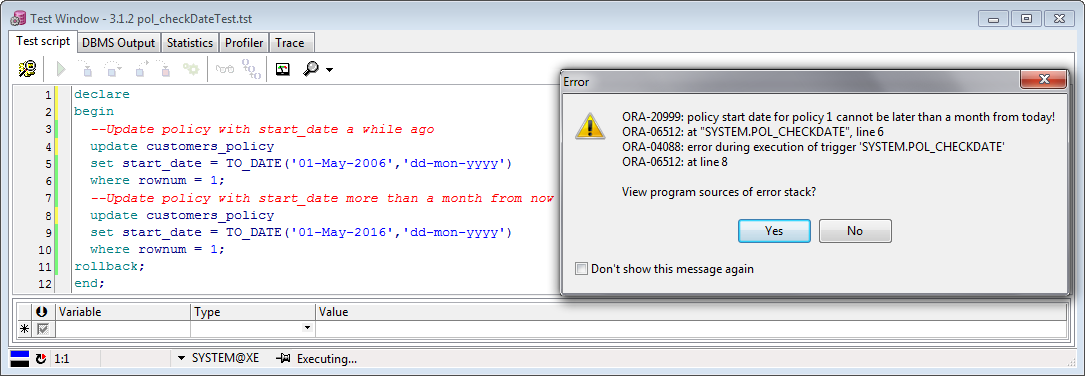
update customers\_policy

set start\_date = TO\_DATE('01-May-2016','dd-mon-yyyy')

where rownum = 1;

end;

Let’s run the above program:



The program complained about the update at line 8, as expected.

**2. cust\_upperLastName** forces the last\_name of each new insert or update into customer to be uppercase, as is often the case in administration offices:

create or replace trigger cust\_upperLastName

before insert or update on customer

for each row

begin

*--Replace last\_name with same string but uppercase characters*

:new.last\_name := upper(:new.last\_name);

end cust\_upperLastName;

This program tests the above trigger by updating the last\_name of the first customer with the name ‘Cohen’ and then outputs the last\_name as it appears in the database after the update.

declare

v\_custLastName\_St VARCHAR2(20);

begin

*--Set first customers's last\_name as 'Cohen'*

update customer

set last\_name = 'Cohen'

where rownum = 1;

*--Get first customers's last\_name and print it*

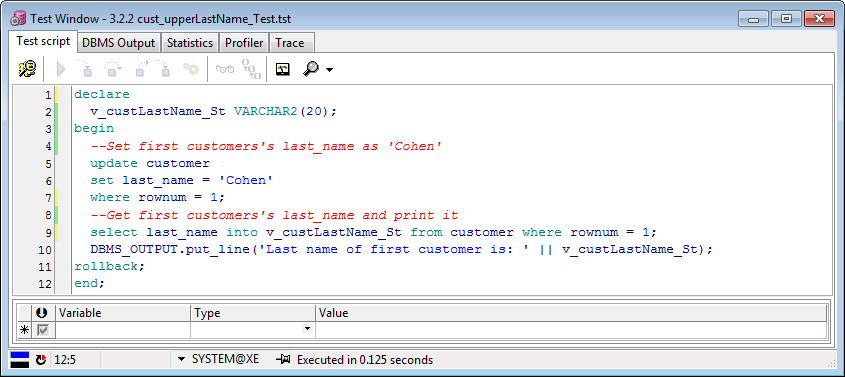
select last\_name into v\_custLastName\_St from customer where rownum = 1;

DBMS\_OUTPUT.put\_line('Last name of first customer is: ' || v\_custLastName\_St);

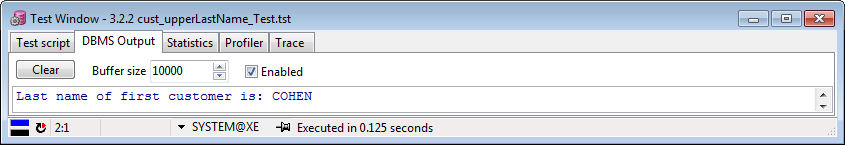
rollback;

end;

Now let us run the above program:



Let’s check the updated last\_name using the DBMS output tab:



Checks out.

**4. Report and Substitution Variables**

**1. Customers report**

This report presents the customer information filtered by minimum or maximum age (we can also define a range this way), by last name and by gender (using an optional list):

select \* from customer

*--Only print records of customers older than this age*

where (sysdate - birth\_date) / 365 > &<name="Minimum age"

hint="Only customers older than this age will be shown" type="integer"

ifempty="0">

*--Only print records of customers younger than this age*

and (sysdate - birth\_date) / 365 < &<name="Maximum age"

hint="Only customers younger than this age will be shown" type="integer"

ifempty="200">

*--Only print records of customers with this exact same name*

and last\_name like &<name="Last name" hint="Enter the customers' last name"

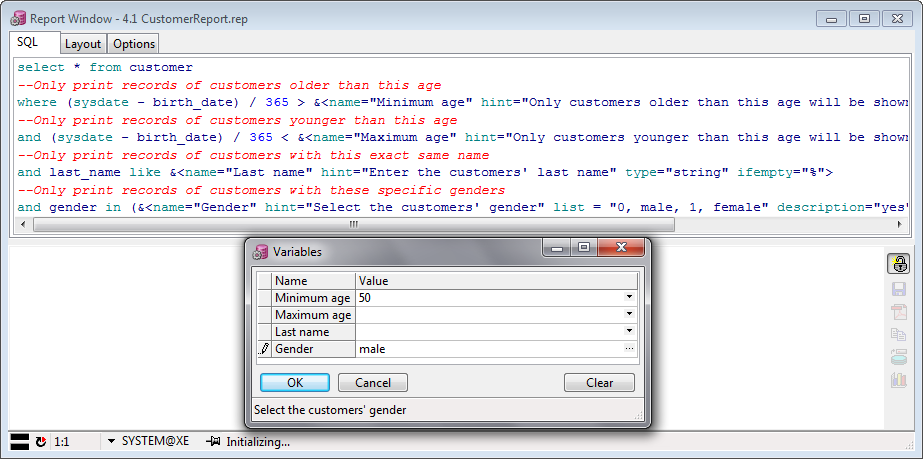
type="string" ifempty="%">

*--Only print records of customers with these specific genders*

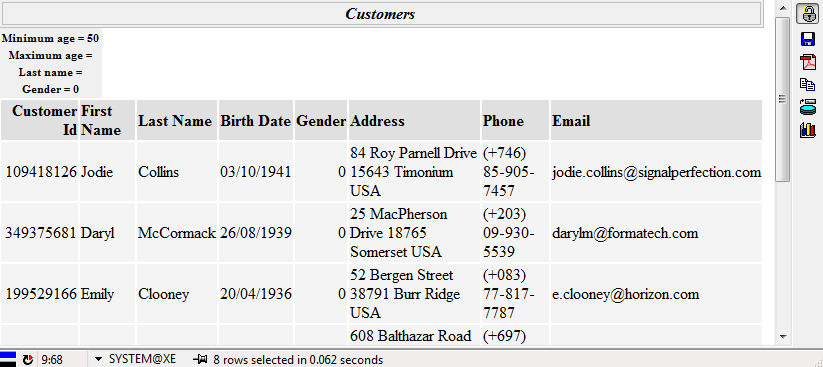
and gender in (&<name="Gender" hint="Select the customers' gender"

list = "0, male, 1, female" description="yes" multiselect="yes" ifempty="0 ,1">)

Let’s run it:



And we get this:



Excellent.

**2. Policies report**

This report presents various information about actual implemented property insurance policies, filtered by minimum start date, age of property and property code (using an optional list). It also offers a layout with ordering and block.

select policy\_id, customer\_id, start\_date, end\_date, price\*(1-0.05\*bonus) as price,

property\_code, age\_of\_property, property\_value

from customers\_policy natural join policy natural join property

*--Only print records starting after this date*

where start\_date > &<name="Starting after"

hint="Only policies that started after this date will be shown"

type="date" ifempty="01/01/1900">

*--Only print records starting before this date*

and age\_of\_property >= &<name="Minimum age"

hint="Only policies for properties older than this number (in years) will be shown"

type="integer" ifempty="0">

*--Only print records with these specific property codes*

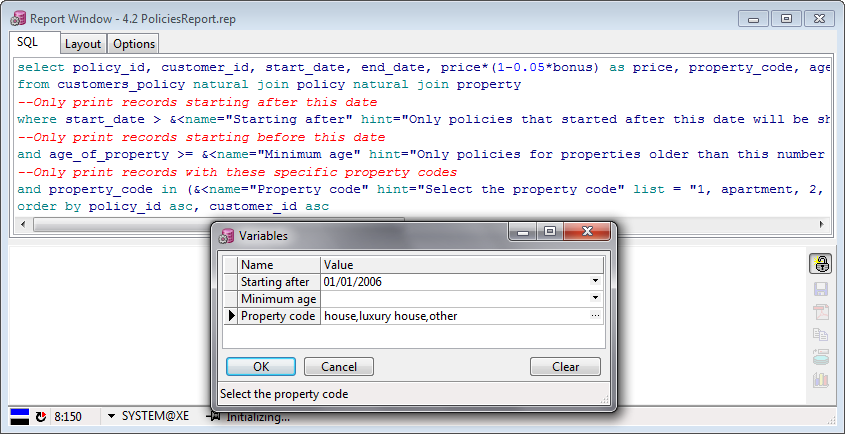
and property\_code in (&<name="Property code" hint="Select the property code"

list = "1, apartment, 2, house, 3, luxury house, 4, building, 5, other"

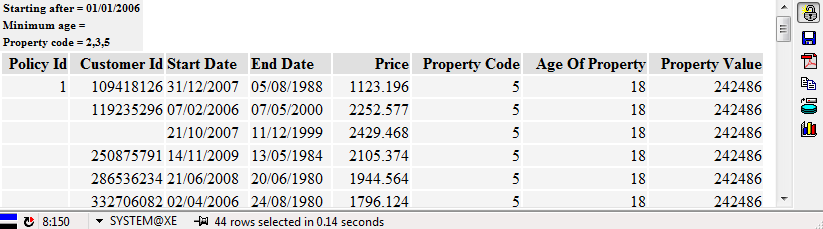
description="yes" multiselect="yes" ifempty="1, 2, 3, 4, 5">)

order by policy\_id asc, customer\_id asc

Let us run it:



And this is the result:



You can notice that we used the layout tab for breaks and tweaked categorization. Perfect.

Level 4 – Graphical User Interface (GUI)

**1. Description**

We wrote a graphical user interface (GUI) in ASP.NET with C#.NET backend code.

For the interface, we used:

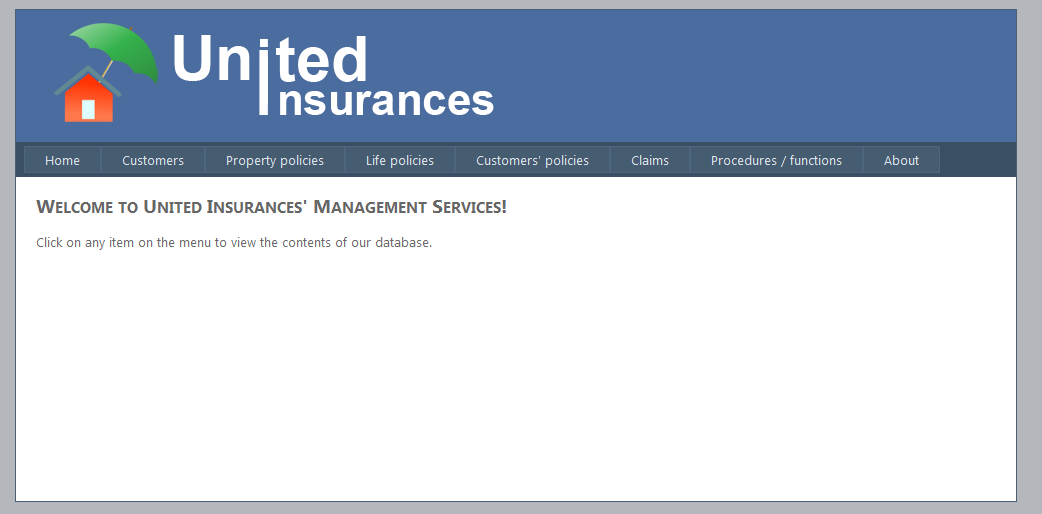
* The default layout with slightly modified CSS and a custom logo made in Photoshop.
* Gridviews for showing the table content and selecting rows, and DetailsViews for editing the row content and handling operations, all binded on SqlDataSources with select, insert, update and delete abilities. These may also be used to demonstrate constraints and triggers.
* Tables, Labels, Textboxes, DropdownLists, a Calendar and Buttons for the form showing the behavior of the function and procedure.

For the backend, we wrote very simple C# with straightforward functions for extremely quick development. This resulted in possible instability, vulnerability (there are two queries for which I have not cleaned the input, exposing us to XSS) and lack of extensibility and reusability.

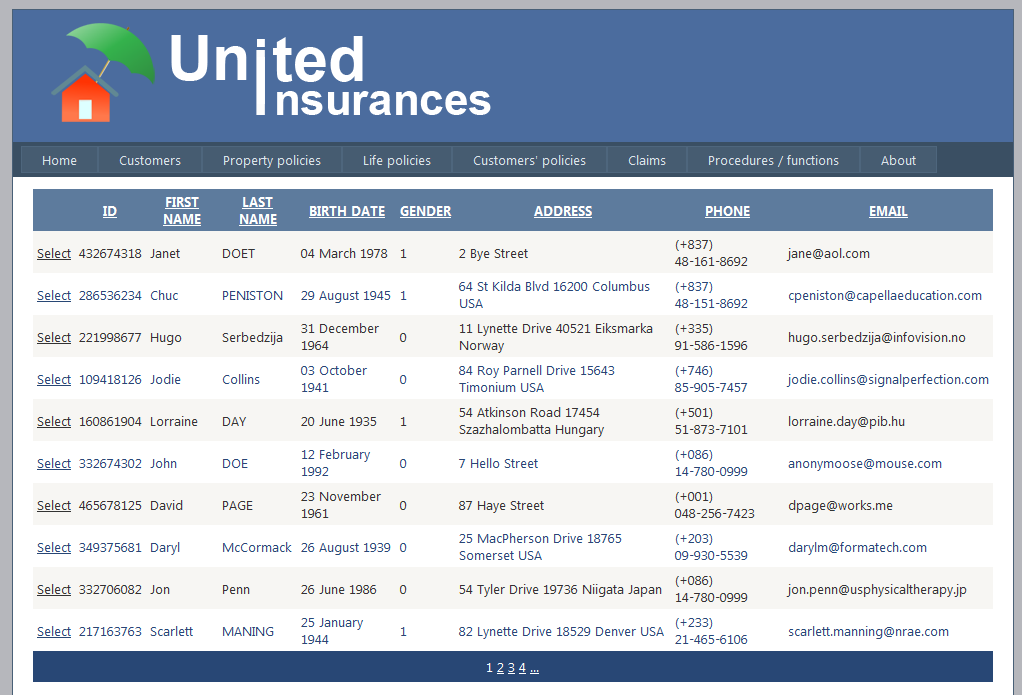
The development was not very easy either (it took three days and quite a bit of headache) but it is relatively reliable, as far as we have tested.

**2. Pictures of each screen**

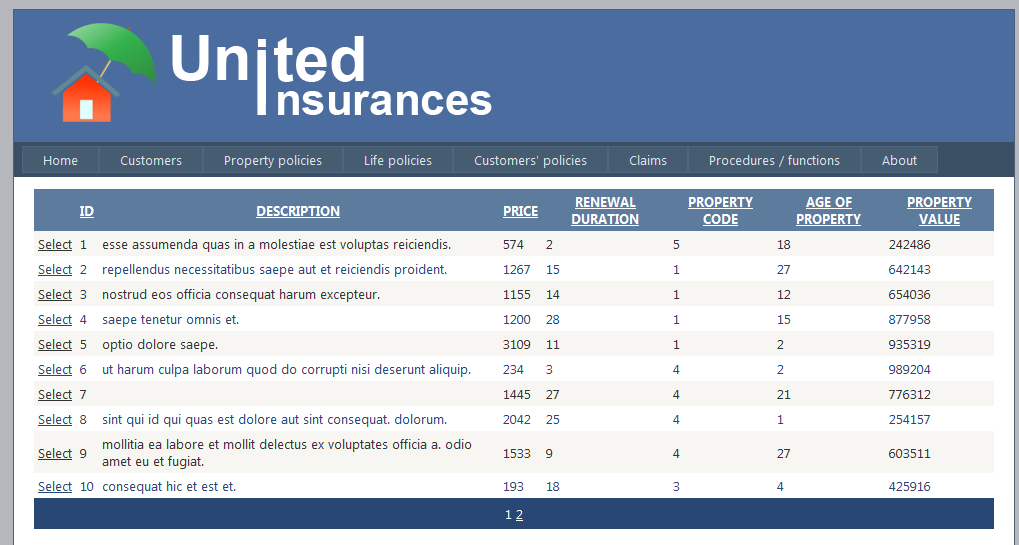
**1. ~/Home.aspx**



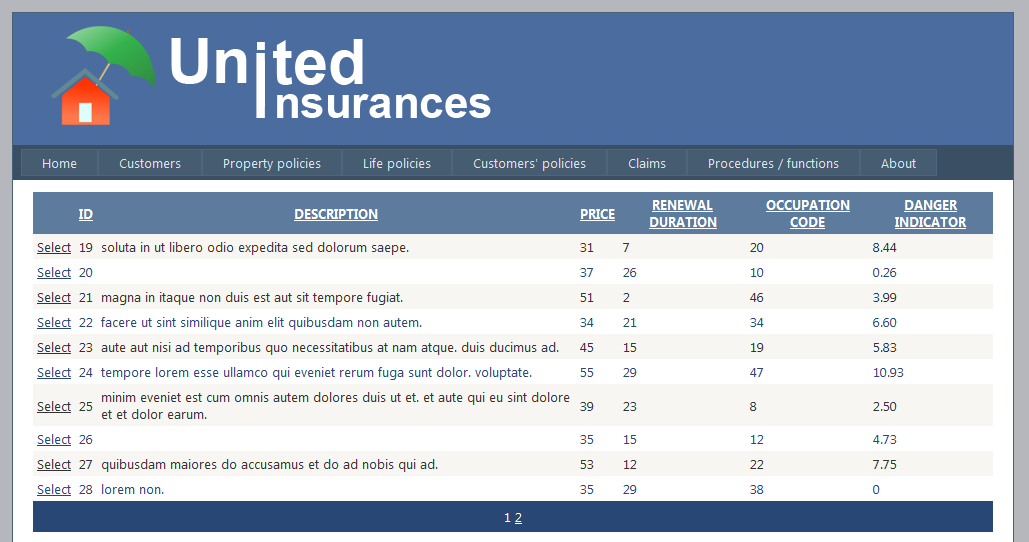
**2. ~/Customers.aspx**



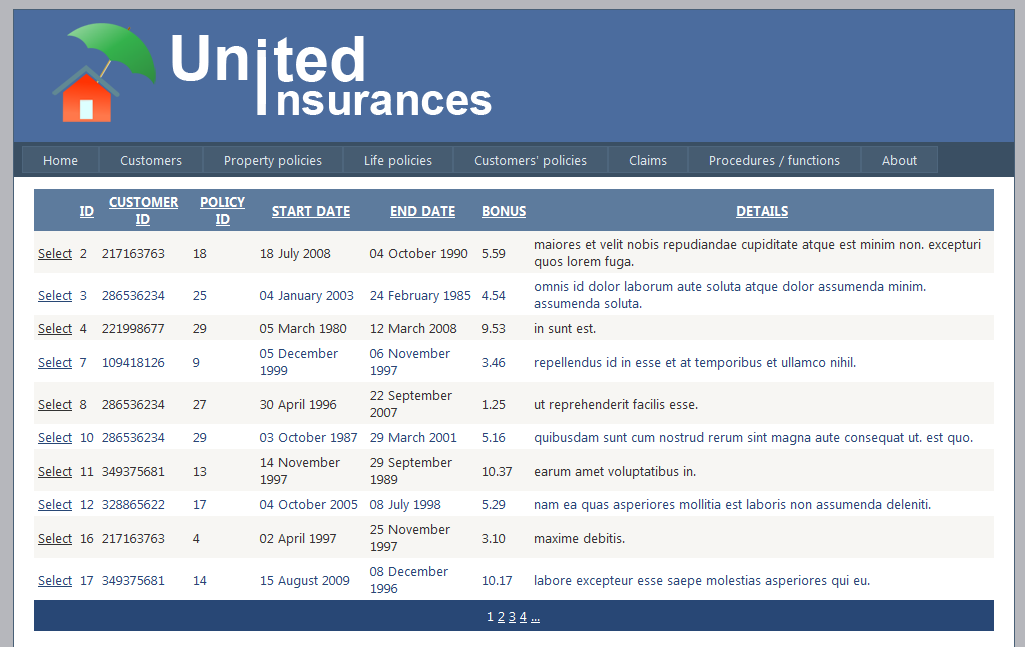
**3. ~/PropertyPolicies.aspx**



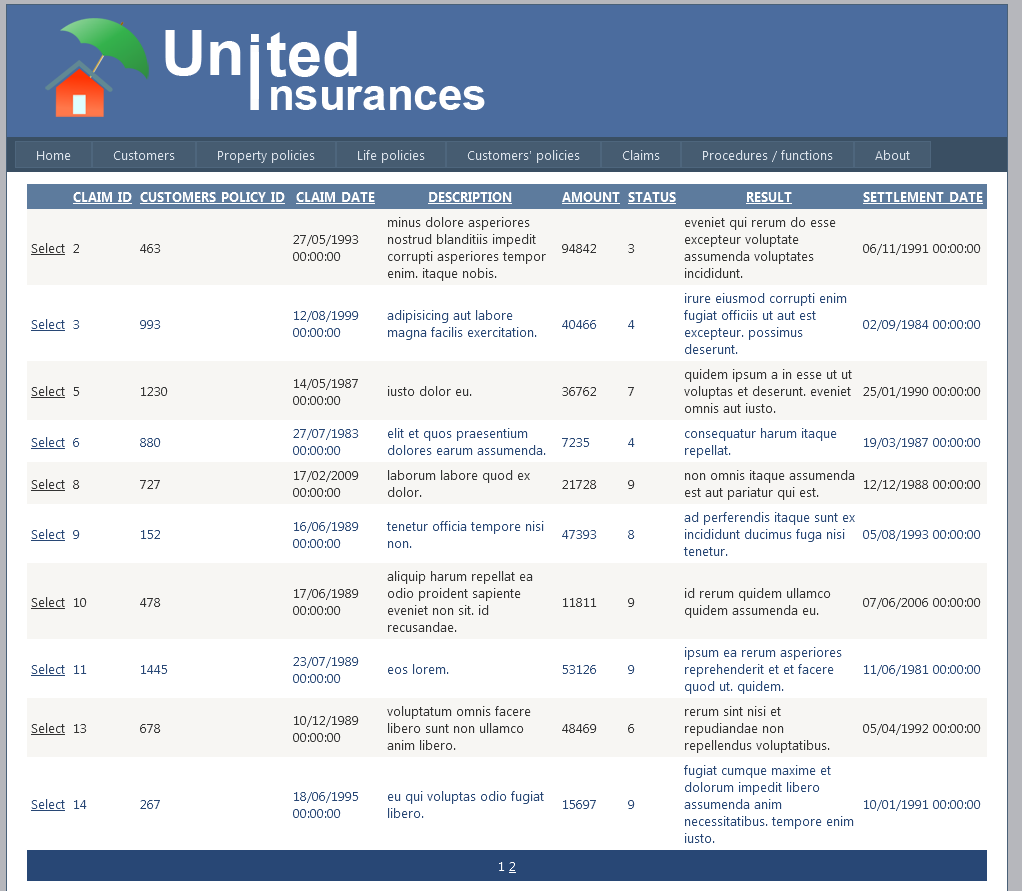
**4. ~/LifePolicies.aspx**



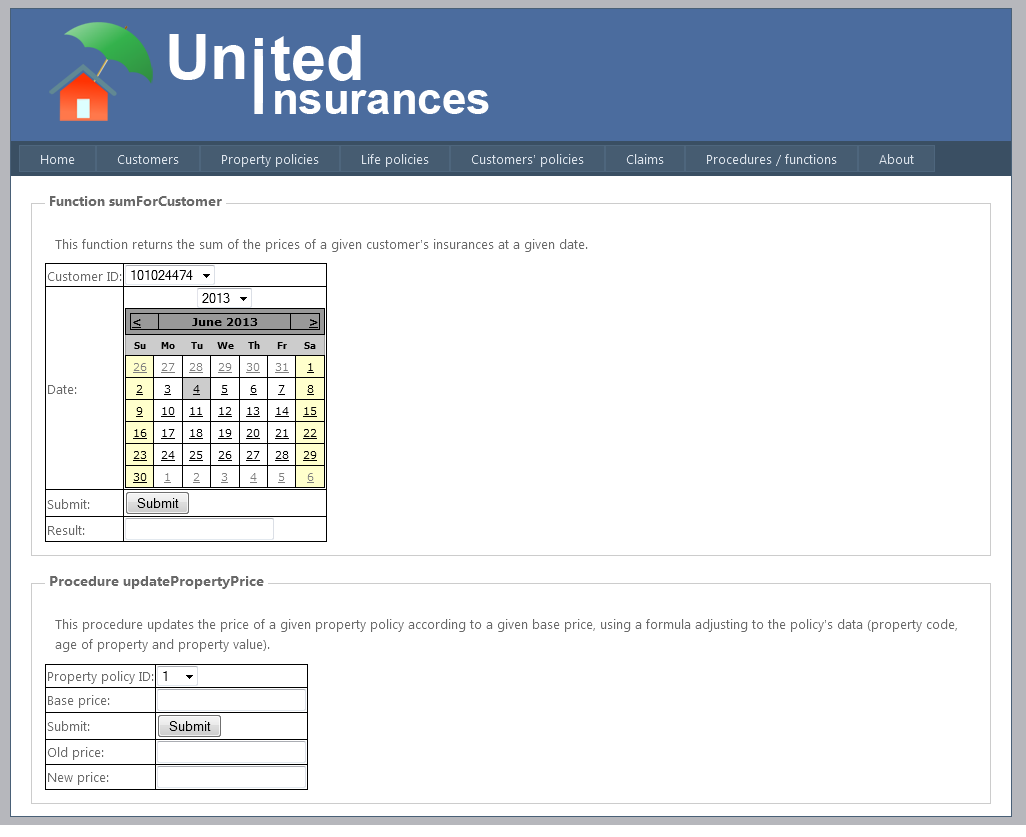
**5. ~/CustomersPolicies.aspx**



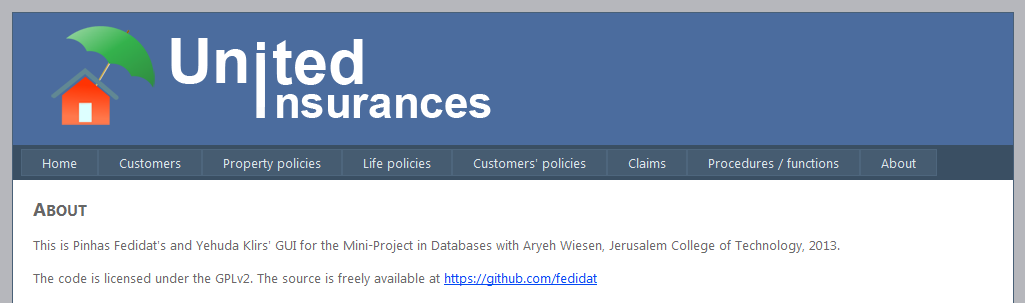
**6. ~/Claims.aspx**



**7. ~/Procfunc.aspx**



**8. "~/About.aspx**



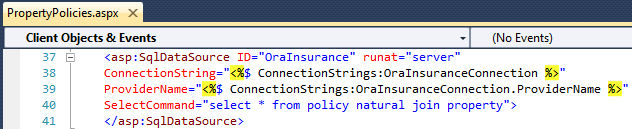
**3. Demonstration**

We will demonstrate the operations SELECT, INSERT, UPDATE, DELETE and the use of the procedure and of the function that we wrote in Level 3

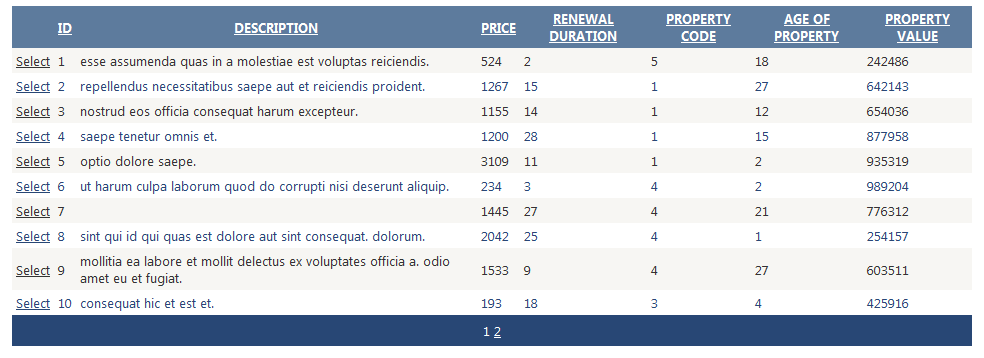
**1. SELECT**

The page ~/PropertyPolicies.aspx shows the result of select \* from policy natural join property

in its GridView (from the sqlDataSource it is binded to, named OraInsurance), as shown below:

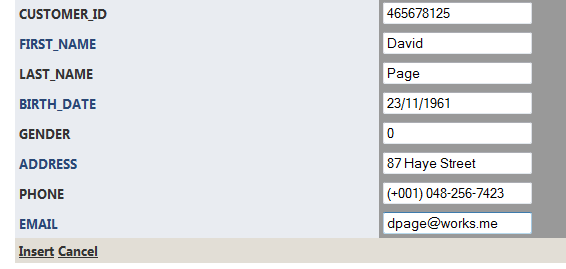


And here is the result on the webpage:

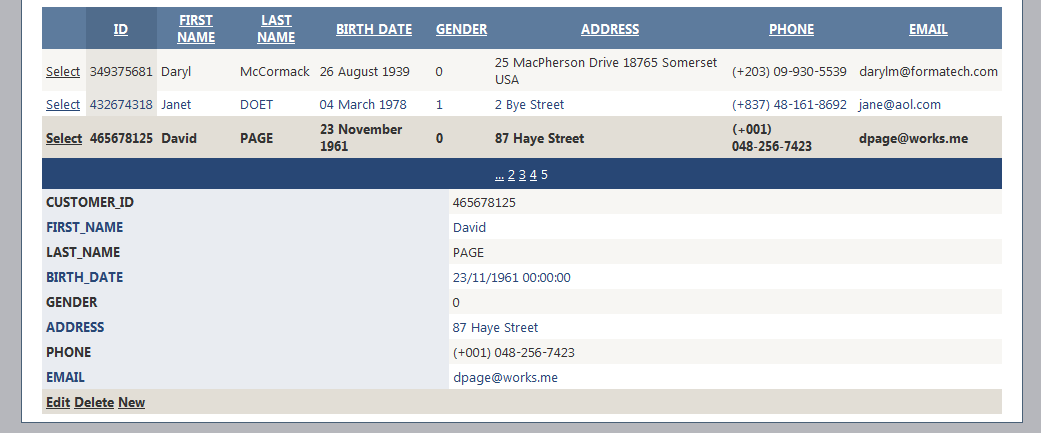


**2. INSERT**

On the page ~/Customers.aspx, we use the DetailsView with the Insert link/button to insert a new customer:



Then we press the Insert link/button and see the result in the table (the new customer is selected in the screen capture):



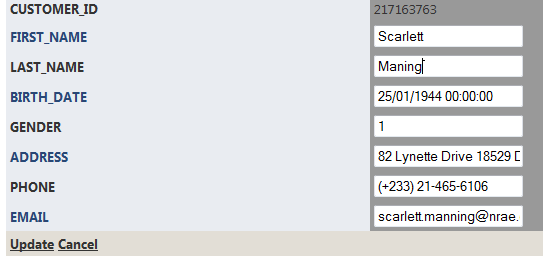
As expected.

**3. UPDATE**

Again on the page ~/Customers.aspx, we select customer ID 217163763:



Then we use the DetailsView with the Edit link/button to update Scarlett’s last name to Maning (in a real world scenario, perhaps this had been a typo that we are now fixing):



Then we click update and look at the updated record in the table:



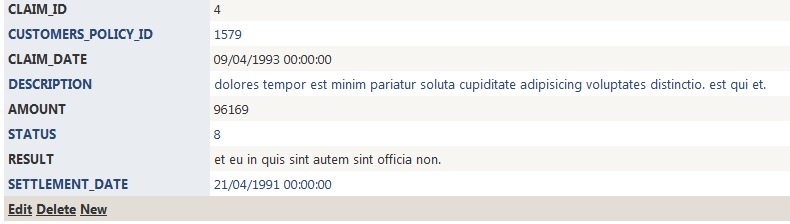
Ah, you must have noticed that Scarlett’s last name has turned to uppercase. This is because of a trigger that we defined in Level 2, which turns the last name of customers on insert or update to uppercase. This is therefore very much expected.

**4. DELETE**

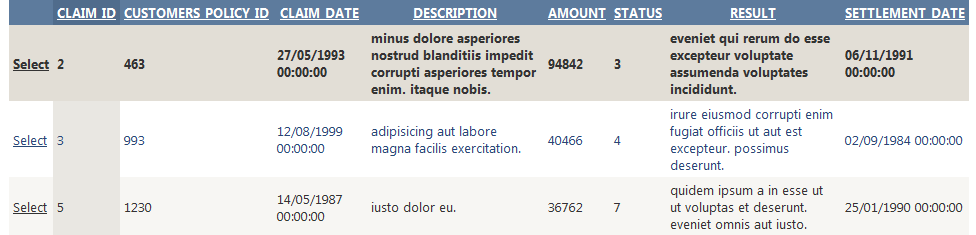
This time, on the page ~/Claims.aspx, we arbitrarily delete claim ID 4, which was applied on policy ID 1579, as we see on the picture below:



Let us select it and press the DELETE link/button:

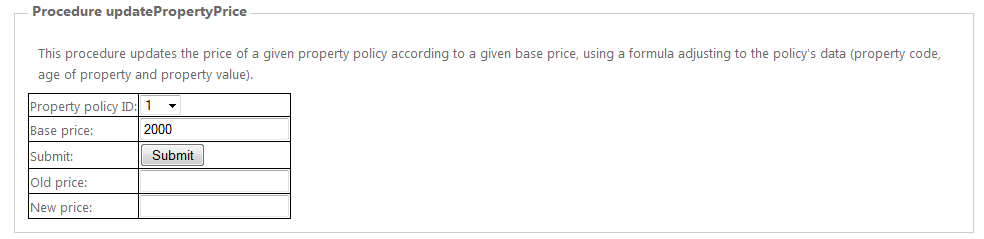


And it now nowhere to be found: when ordered by ascending ID, we see the table going from ID 2 to 3:

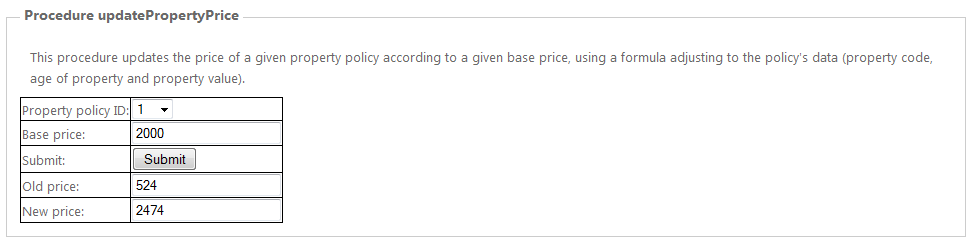


**5. PROCEDURE**

On the page ~/ProcFunc.aspx, we go to the “Procedure updatePropertyPrice” panel we arbitrarily select property policy ID 1 and enter 1000 as the base price for the new price of the selected policy.



We now just have to press Submit and see the result:



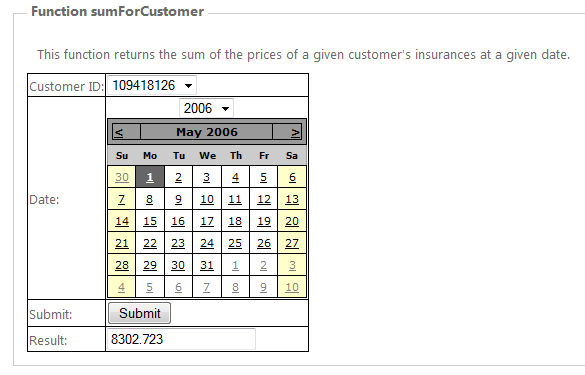
We can check in ~/PropertyPolicies.aspx that this is indeed true: below are the comparisons of before/after the procedure execution:





**6. FUNCTION**

Finally, again on the page ~/ProcFunc.aspx, we go to the “Function sumForCustomer” panel we choose to get the sum of policy prices for the customer ID 109418126, then on the DropdownList we select year 2006, the Calendar automatically moves to year 2006 (we implemented this because pressing 7 times the back arrow on the Calendar can be tiring), then we select May 1st, and finally we submit:



We have already proven the accuracy of this function in Level 2 and despite the fact that the policy prices have been modified since then, the function has not, and remains valid. We can therefore trust this result.

**4. Conclusion**

In retrospect, this project gave us the occasion to look at a lot of applications, methods and concepts:

* Model conception of RDBMS using DDS-Lite v2.21 with entities, relationships attributes, keys, normal form.
* SQL queries on an Oracle 11g express database with many types of operators.
* Writing views, indexes, commit/rollback, constraints, triggers, PL procedures, PL functions, PL programs and reports with substitution variables on PL/SQL developer v10.
* Connecting an Oracle database to a Java program from Netbeans or a C# program from Visual Studio 2010, as well as developing an ASP.NET GUI based on C# in a few days from very little knowledge.

On a personal note, this was also the first truly organized project for which we used a Control Versioning System (in this case, git, with a Github remote repository). At the time of writing, it is publically available on <https://github.com/fedidat/Mini-project_in_Databases>. All code written by us is licensed under the GPLv2.

We would like to thank Aryeh Wiesen for helping us along the way on the few occasions during which we struggled.