

# The York Family Genealogical Database

The goal of this project is to design and implement a genealogical database of our extended family. This database proves itself useful to not only the immediately family whose genealogy was traced but also those future descendants who might want to trace their lineage. The goal of this project was to create a working model for a real-time database system. In order to manage our data, we were given an Object-Oriented Database Management System (OODBMS) called ObjectStore. It consists of the ObjectStore Java Interface Release 1.3 (1998-04-14 13:18:44 on Java) and the ObjectStore Release 5.1 for Solaris 2.x (SunOS 5.x) Sparc/SunPro. We will take a look at ObjectStore now.

## OBJECTSTORE

This information was taken from ObjectStore's homepage. "ObjectStore Enterprise Edition is the leading object database management solution for developers creating dynamic, reliable, high-performance applications for the Internet and other distributed computing environments...It provides the best of breed, object-oriented database binded directly with Java, C++ and ActiveX to enable the development and delivery of high-speed, complex web transactions, dynamic content, and network management applications for today's enterprise...ObjectStore Enterprise Edition provides a distributed multi-tiered

architecture that leverages replication and caching of both data and services to meet unprecedented scalability demands. Distributed caches provide local, in-memory data access to the various services and components that make up the application, and both data and processing are distributed across each application tier. This cuts down on repetitive and unnecessary network traffic, removes database server bottlenecks, and allows scaling without adding expensive hardware... ObjectStore's Enterprise Edition provides a full set of reliability and high availability options for delivering full 24x7 continuous operation that ensures the integrity and reliability of your applications most valuable resource: it's data."<sup>1</sup>

An object-oriented database provides advantages over the traditional Relational Database Management System (RDBMS). A RDBMS won't let you recursively query but the OODBMS will. An OODBMS allows one to store complex attributes and types as well as multimedia datatypes. This multimedia has to be accessed as a Binary Large Object (BLOB) in a RDBMS which means you can't access certain portions of movies or sound clips inside that file; you have to retrieve the entire BLOB first. ObjectStore allows one to store and modify the data as a persistent object. If data is persistence-capable, then that just means that it can be stored in a database. A persistent object is a representation of an object that is stored

---

<sup>1</sup> <http://www.objectdesign.com>

in a database. After an application retrieves an object from the database, the application works with the persistent object in the Java environment.

I chose the Java API (Application Programming Interface) with Java 1.1.6 JDK because using Java would allow me to create a decent GUI (Graphical User Interface). It would be much easier than trying to create a GUI with the other alternative, C++. ObjectDesign did a wonderful job providing documentation on their product ObjectStore. If more students had read the documentation, there would have been fewer problems. The reference guide was good but there could have been more examples of code in their documentation. After the design phase, a few adjustments needed to be made which required me to present a new design model. You can find the modified Object-Oriented Entity Relationship Diagram in Appendix A.

### DESIGN PHASE

In the restructured ER Diagram, I found it easier to implement one "Individual" entity and two relationships "Spouses" and "Children". The Individual entity consisted of 12 attributes:

- name (a name, type String)
- address (an address, type String)
- pob (a place of birth, type String)

- dob (a date of birth, type String)
- dod (a date of death, type String)
- gender (gender, type String, "Male" or "Female")
- id (primary key, an identification, type String)
- age (an age, type int)
- mother (mother's id, type String)
- father (father's id, type String)
- spouseVec (a dynamic array of spouse ids, each element of type String)

I supplanted three other entities by placing their attributes under the Individual entity. I did this for the simple reason that they were impractical in an OODBMS.

I also saved space on storage somewhat by only storing one object for each individual. I used a String (their id) in the other places that referenced the object. This String helped me navigate through a hashtable I used to find the person's object. I also discovered that I didn't need an array of "Children" as well because I decided to store a "mother and father" attribute. The "Children" relationship could be derived.

## IMPLEMENTATION

There are a few rules and integrity constraints one must pay attention to when creating and modifying the database:

- The spouse relationship should be many-to-many. In my project, any individual could have a Vector (dynamic array) of as many spouses as they wanted.
- The parent-child relationship should be one to many; that is, one parent can have many children. This is naturally enforced in my database.
- If someone tries to add an id to the database that already exists, give an error message and abort the transaction.



- Date of death of an individual should be later than date of birth
- Date of birth of individual is related to life span of his/her parents
- A person should not be married to two individuals at the same time (but this may be culture dependent)
- Children of same woman must have certain age difference

- Date of marriage must be during the life span of individual, with possible additional restrictions (depending on their culture)
- A person cannot be its own parent or parent of a parent, etc.
- Biological parents of an individual cannot be of same gender

If a database system were to be foolproof, then all of these constraints would be implemented. They sound easy to implement but the truth is that some of them are very costly in terms of processing time because of several nested loops needed to check these constraints. I liken this to the example of including "DISTINCT" on SELECT statements in SQL; it's very costly!

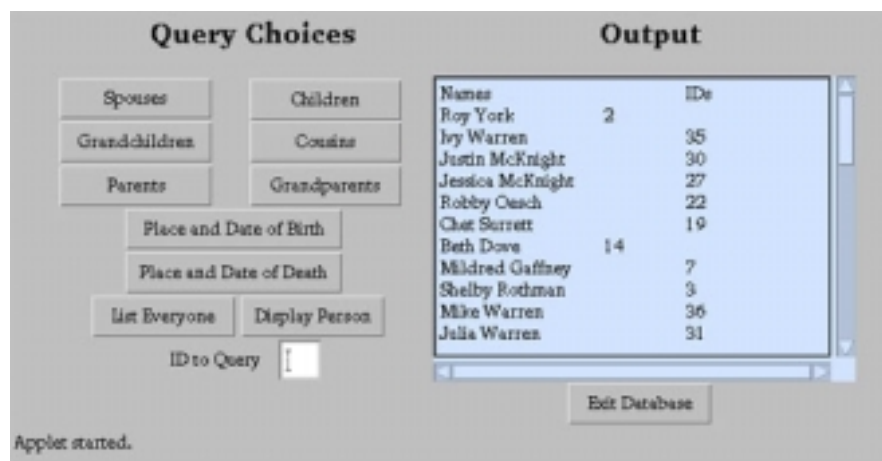
The queries are fun to write but I find myself duplicating many lines of code simply for the fact that each method of mine starts out with a transaction read statement. If I call another method from within the current method, it throws me an exception because you can't have two transactions open at once. That is something that will have to be worked around somehow in the future. I'm going to review the 10 queries I implemented as well as give you a screenshot proving their success. A family tree of the individuals and their respective ID numbers are located in Appendix B for your perusal.

## QUERIES

The "List Everyone" query is passed a TextArea which will contain the results of the query. A read-only transaction is started and the hash is enumerated so as to be able to go through every person in the database. In my screen shot below, the numbers that appeared non-justified in the output mystified me. I checked for those individuals output in other queries and they were the same way. I tracked it down to the fact that I accidentally entered "backspace characters" when I was adding some of the individuals to the database, and that backspace character was getting interpreted within the string. The algorithm is as follows:

For every person in the database

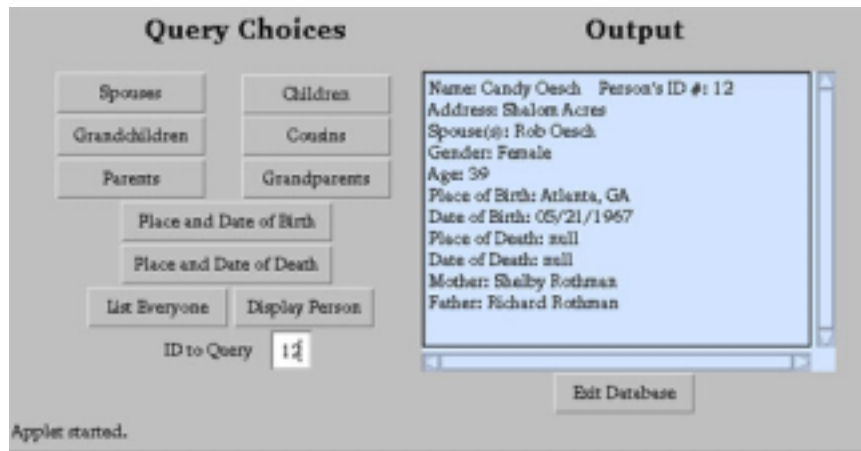
Append their name and id to the TextArea.



The "Display Person" query is passed a TextArea and the string of the person to show. The hashtable, which is stored in the root node of the database, is enumerated once again and the algorithm is simply:

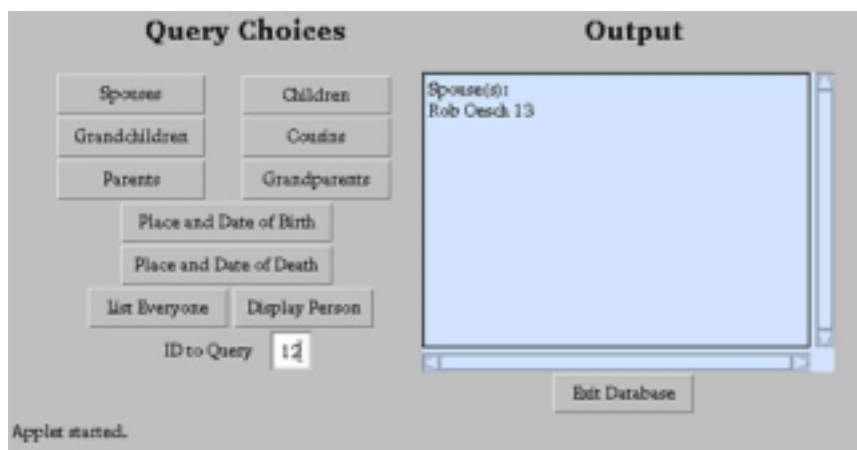
For every person in the database

Append all their attributes to the TextArea.

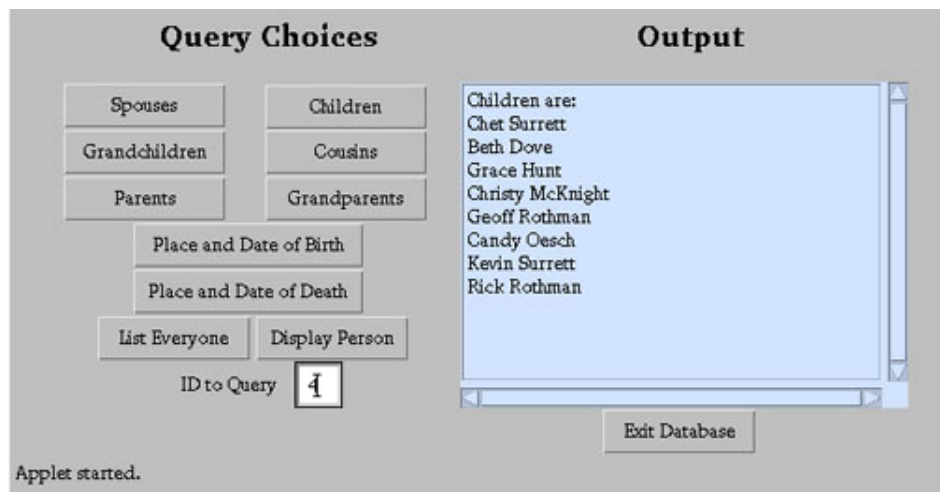


In the "Spouses" query, you pass in a TextArea and the id of the person whose spouse you want to find. The vector is first searched to see if it even contains a spouse or not. If not, an error message is generated and the transaction aborts.

The method makes an enumeration of all the String ids in the vector, and for each id, prints out the person's name and id.

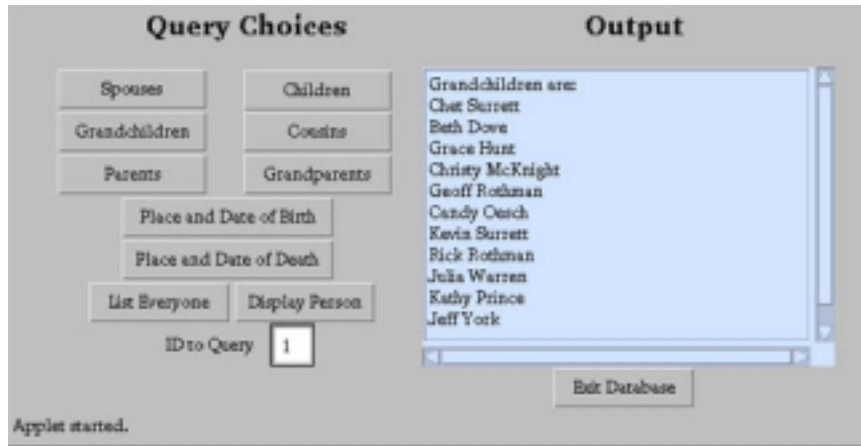


In the "List Children" method, all the people in the database are enumerated, and I set a boolean called "found" to false (a child not found). Since there are no objects that explicitly store the children, this relationship must be derived. While we search through everyone in the database if the current person has a mother id field or father id field that matches the one we passed in, we know we've just found their child so we can print the child out.

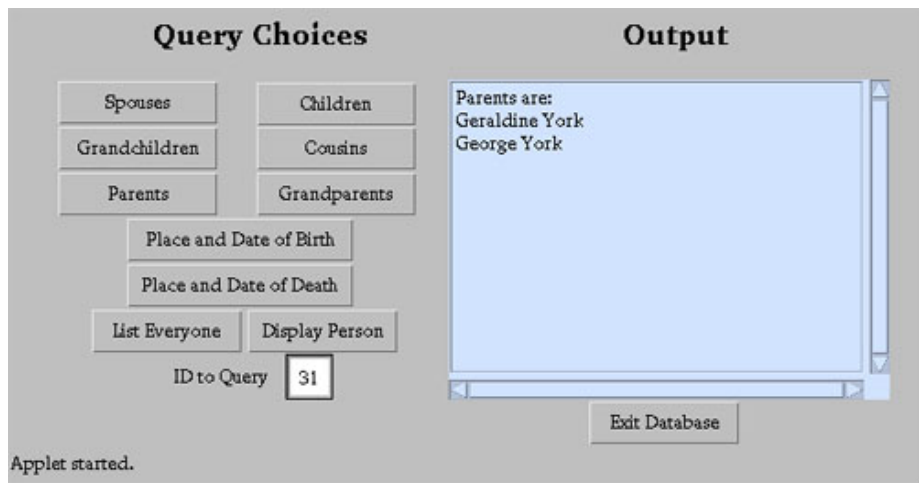


Next is the "List Grandchildren" method where we get an enumeration of everyone and set the found boolean to false. . While we search through everyone in the database if the current person has a mother id field or father id field that matches the one we passed in, we know we've just found their child. You call the searchPerson method and return the object of the child you just found. You start the process over and search for a person who lists that child as their parent

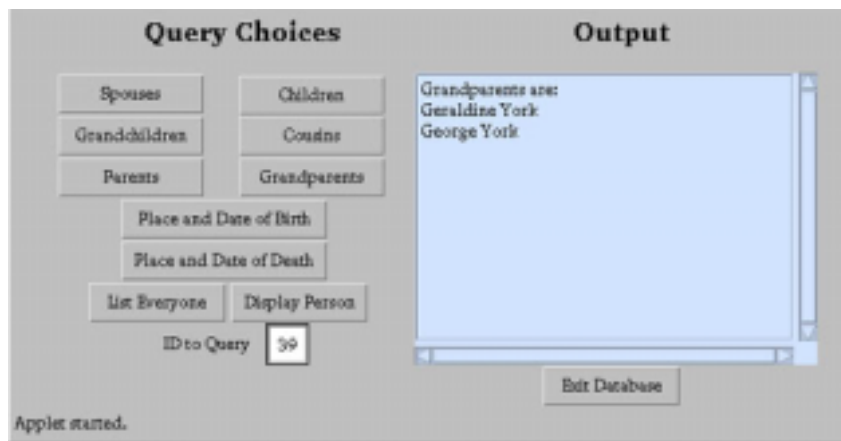
(motherid or fatherid). Recursion would've been pretty here!! (bad transaction problem, bad!!) ☺



"Listing the Parents" proved to be much easier. You pass in a TextArea and a child id whose parents you want to find. You get the child's record straight from the hash and get his mother and father's id. Then you call the searchPerson method for both of them, in order to print out his and her name in the TextArea. Voila!



For the "List Grandparents" query, pass in a TextArea and the id of the child whose grandparents you wish to find. If the child has a mother or father (which they will unless there's a mistake, they're a spouse, or they're the top of the genealogical tree) then get their mother and father's ids (your grandparents). Use the searchPerson method to return the objects of your grandparents and then print them to the TextArea.



The "List Cousins" query was just plain nasty and would've benefited from recursion greatly (NB problem above). The algorithm is as follows:

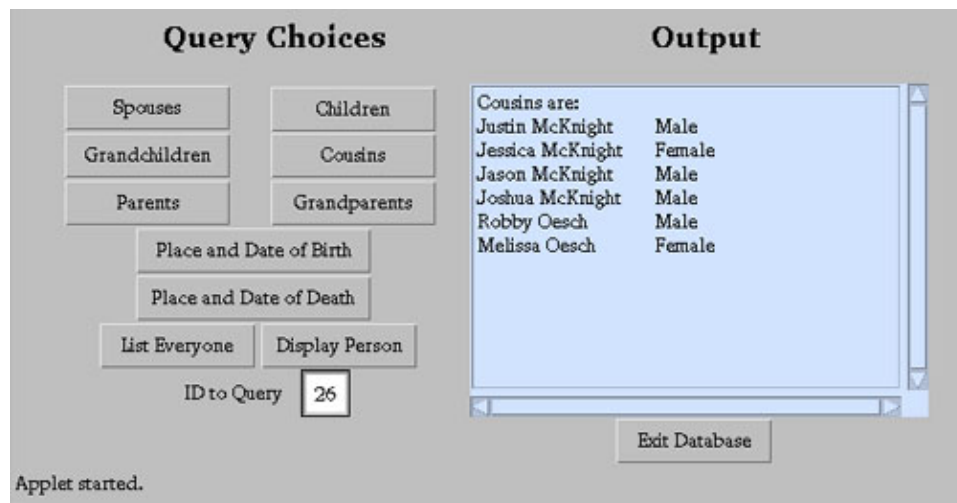
Get the object of the child who was passed in. Initialize two strings that will retain the values of their parents String id.

If the child has a mother or father, initialize the above two strings appropriately.

If their mother's motherid field is not null

pass grandma and your mom's ID to the "listGrandchild2" method

The listGrandchild2 method is the same as the listGrandchild method except the transactions and a couple of other things were removed to avoid conflicts. The above method is passed the four grandparents of the child. You use the "childMom" and "childDad" id to prevent your grandparent from trying to find your parents children (you and your siblings). This is Kentucky but still "you are not your own cousins!!!) ☺



## CONCLUSIONS

The project was pretty self-explanatory. We were given the task of implementing a genealogical family tree in an OODBMS called ObjectStore. We explored using persistent objects and classes. I benefited two-fold by learning both Java and an OODBMS simultaneously. Initially, I didn't expect to spend around 100 hours on this project but I did.