The major components of such a system are:

- Set of Database/Query Objects (DO/QO) are C++ class representation of individual tables and joins. The join conditions are deduced from Foreign Key relations. Developers may provide additional join conditions, if required.
- SQL Class (CC) provides the interface for creating runtime objects for database interaction.
- Business Logic Level (BLL) makes it easy to rapidly write the Business Logic.
- Code Generator. A set of tools for generating C++ Database/Query objects.

A Query Object Layer (QOL) provides the framework for creating runtime Objects for database interaction. The rest of tasks to interact with the database, query generation, execution and etc. are all handled by the QOL and BLL. Logic Layer performs the desired operations.

The turn around time to completely incorporate all the schema changes is short. The Objects that hold the database schema information (rows and tables) are generated by the code generator. The developer does not need to know about the schema as such. This layer provides simple insert and select interfaces on the C++ objects (generated table and view). The interface implementation is templated and can be used for any generated row or table objects. The template interface gives a good flexibility to incorporate any changes in the schema, since the objects can be regenerated by the code generator and the interface implementation remains the same.

The Organization of this layer is shown in the Figure 2.

- *RowClass* is the generated C++ object (class) corresponding to a row in a specific table. The RowClass objects inherits from a parent class RowInterface and implements the interface for simple setValue and getValue calls to set or get the values to and from the RowClass.
- *SchemaConstraintClass* holds the entire schema including the foreign keys, unique keys and other references in a table. This class is generated, so any changes in the schema will get reflected in this class. It inherits from a parent class BasicSchemaConstraintClass, and implements the interface for simple getSchema, getPrimaryKeys, etc calls to get the schema information about the table.
- *SQLClass* can generate the SQL statements for any table from its the schema. Its implementation is independent of the changes in the schema.
- *TableFrame* is a templated container that can contain a vector of RowClass. It inherits from TableFrame class, and implements the interface for insert and select calls. It instantiates objects of SchemaConstraintClass, RowClass and SQLClass to contain the row data, the schema for the table and the mechanisms to generate the SQL queries.

The user interaction with the application server is either through a Web Services Interface or through a Python Interface Layer.

Thanks to: Burt Holzman, Sinsenia Veselli (Fermi Lab), Peter Elmer and Laszlo Tura (CEPNI).

The Dataset Bookkeeping Service prototype is an example of how to incorporate the QOL into the Bookkeeping System. The prototype is implemented in C++ using the Boost Container library and the Python interface. The QOL is configured by the user. It provides a simple insert and select interface. The Business Logic Layer provides implementations corresponding to API calls. These implementations are called Managers which instantiate the objects from SQL and make use of the interface provided to retrieve or insert data. Any logic about validation or cross referencing between various table and multi tables are done in this layer.

The whole development cycle requires only:

- Describing QOL and running the Code Generator to produce Database and Query Objects.
- Writing appropriate Client Data Structures and their translation into OQL.
- Writing Business Logic, which is mainly the instantiation of appropriate OQL Database/Query Objects and invoking API calls.

The rest of tasks to interact with the database, query generation, execution and etc. are all handled by the QOL and BLL.

The Advantages:

- There are fewer chances of making any Query mistakes. We have several examples where a failure in OQL actually pointed towards a mistake in the DDL and Schema.
- The turn around time to completely incorporate all the schema changes is short.
- The chances of executing an incorrect query are minimal as query generation is done automatically.
- The user interface deals in terms of views (of single and multi tables) and rows which is much simpler than dealing with queries.
- The development cycle is simple and short since most of the code is generated by the code generator.
- The plug-in has a simple interface and can be easily used in any application.

**For each Table in MultiTable view**

- *tableName*
- *dataFrom DB if exists*
- *in Primary key or lookup key only*

**Insert Algorithm**

- *insertRowInterface*
- *sql statement*
- *generate SQL and execute*
- *write into C++ Table and Rows object (using SWIG) and invoking API from Interface layer.*

**Generate and Execute Insert statements**

**Set the Foreign Keys in MultiTable view**

- *sequence of foreign keys in MultiTable view*
- *SQL statements to set*
- *store key in foreign table*