Introduction

In this article we present the Z++ abstractions for the marriage of relational database and object-orientation. A happy marriage necessitated the compromise of confining Z++ to the Data Manipulation Language (DML) of SQL standard.

The Data Definition Language (DDL) deals with system-level operations, relatively speaking. These operations are generally performed by system administrators who utilize specialized interfacing programs provided by vendors of database systems.

An application, on the other hand, needs to access or manipulate the data in an existing database. We are using the term application relative to a database system. In that sense an application should use an existing database, not to create one, nor to modify the schema of an existing database.

The design of language should be based on a verifiable conceptual model. In the next section we introduce the Z++ model for database statements. In order to simplify the discussion, we review a few details that Z++ standard library automates.

Preliminaries

Database operations require establishing a session, which includes connecting to the server and logging in, and at some point ending the session. These operations have been abstracted away through declaring a database object. At elaboration a session is established, which terminates when the object goes out of scope.

The Z++ type of database object is defined in the system include file database.h. All operations are implemented in the standard static library. However, beyond a plain declaration, there is no need to be aware of the details of the methods that the database system type defines. All methods are used internally by, the Z++ compiler.

In the statements presented in the next section, a database-object refers to an instance of database class in the include file database.h, that one declares before interacting with a database system. The declarations require standard arguments, such as user-name and password, as shown below.

```
databaseType Dbase(IP-address, port, database-name);
databaseUserType Duser(Dbase, user-name, password);
```

The first declaration provides the data needed for the second declaration. It is the second declaration of type `databaseUserType`, which the Z++ database statements use. A session begins with the second declaration, as well.

The Model

Z++ is an object-oriented superset of C++, which supports templates for defining containers such as list abstract data type. Prior to its use, a template must be instantiated

with a specific type. In this article we will refer to the type of object instantiating a template as the catalyzer.

The purpose of executing the SQL select statement is to receive a set of records from a database. A natural way to manipulate the set of records obtained from a query is to copy them to an instance of catalyzer and insert them into the container. Let us refer to the process of copying a record to an instance of a catalyzer (and vice versa) as mapping.

The Z++ select statement combines the following: full support for an SQL query, the specification of a template and a catalyzer, and a mapping of database fields to members of the catalyzer. It further allows specifying the container method for inserting instances of catalyzer into the container.

The basic model as described for the select statement remains the same for other Z++ SQL statements, presented next.

The Select Statement

The syntax of Z++ select statement is similar to the SQL select statement. Keywords are shown in blue. All punctuation is part of the syntax, except the meta-symbols []. The brackets [] indicate zero or more occurrences (the Kleene star operator).

```
databaseSelect<user_object, catalyzer_type : table_name [, table_name]>
    table_name.field_name<catalyzer_member>
    [,table_name.field_name<catalyzer_member>]
    where (boolean_expression)
    using conatainer_instance, container_method
;
```

The user_object is an instance of type `databaseUserType`, which we discussed earlier. The catalyzer_type is type of a Z++ object used in mapping. The mapping is defined via the expressions “table_name.field_name<catalyzer_member>”, where table_name is name of one of the tables listed after the colon.

Note that, names of tables and fields are those that one would directly submit to an SQL statement. In other words, the strings are not Z++ identifiers that evaluate to those names. Rather, they are the names.

The expression for “where” can contain mixture of database fields and Z++ objects. Literals can be used directly. However, a Z++ expression must be enclosed between {}. The expression, of any complexity, will be evaluated and passed to the database at run time. The type of expression must be one of Z++ fundamental types of numeric or string.

The arguments to the using phrase are an instance of container to be populated, and the method of the container template to be used for populating it. The container is to be instantiated with the type of the catalyzer.

The semantics is that, upon the execution of the select statement, the result of the query will populate the container-instance using the specified container method, and the mapping defined via the catalyzer.

Other statements

The remaining statements are syntactically analogous to the select statement. However, unlike select statement, they are generally intended for use in a loop. Note that the select statement does not require a loop.

The statements resemble, and behave like their corresponding SQL statements. Below is the insert statement. The compiler transforms the following to an SQL insert for the catalyzer-object in the using phrase of the statement. Thus, in a loop where the catalyzer is ranging over objects in a container, one can simply check on the state of the catalyzer in order to decide whether or not to execute the insert statement on it.

```
databaseInsert<user_object, catalyzer_type : table_name>
    table_name.field_name<catalyzer_member>
    [,table_name.field_name<catalyzer_member>]
    using catalyzer_object
;
```

The remove statement is similar to the insert statement, except it allows a where-expression, which is sent to the database for evaluation. As in the case of insert, one can also check on the values of members of catalyzer in order to decide whether or not to execute the remove statement. This applies to update statement, as well.

Note that remove and update do not have a using clause. In case of insert, the statement uses the catalyzer object for inserting a row, which is why it has a using phrase.

```
databaseRemove<user_object, catalyzer_type : table_name>
    table_name.field_name<catalyzer_member>
    [,table_name.field_name<catalyzer_member>]
    where (boolean_expression)
;
```

The update statement includes a set-expression, which it sends to the database. A set-expression can include Z++ expressions just like the where-expression. The syntax of set expression is the same as SQL set. Z++ expressions for values of fields need to be enclosed in curly brackets {}, as they do for where-expression. Literal values can be used directly, without the use of {}.

```
databaseUpdate<user_object, catalyzer_type : table_name>
    table_name.field_name<catalyzer_member>
    [,table_name.field_name<catalyzer_member>]
    set set-expression
    where (boolean_expression)
;
```

Fetch Size

The constructor of `databaseUserType` has two optional parameters: an integer for size of fetch, and a string for IP-address of a database proxy.

The select statement uses the fetch size for an initial fetch. If this parameter is 0 (the default value), select will fetch the entire set of rows and insert them in the container object. For any other positive integer n, select will only fetch n rows and insert them in container. The remaining rows can be fetched using the fetch expression, shown below. Fetch expression is identical to select statement except for lacking the where-clause.

```
databaseFetch<user_object, catalyzer_type : table_name [, table_name]>
    table_name.field_name<catalyzer_member>
    [,table_name.field_name<catalyzer_member>]
    using conatainer_instance, container_method
;
```

The fetch expression returns true so long as there are more rows to fetch. On each execution the fetch expression will fetch n rows where n is the value of parameter for fetch size, or fewer if the result of query has fewer rows left.

If all rows in the result of a query are fetched, the Z47 processor automatically performs the necessary cleanup. However, when the fetch expression is used, it is likely that one may choose to stop fetching before all rows are received. In that case it is necessary to do the cleanup manually. The database free statement is quite simple and does just that.

```
databaseFree<user_object>
```

The single argument to the database free statement is the same as the first argument to all other database statements.

Database Proxy

The Z++ Internet Server includes a proxy for database operations, among other services for distributed computing. The proxy is useful for mobile devices. However, some desktop applications may also utilize it for efficiency.

The important fact about the design is that the Z++ application using database statements does not need to be modified when switching back and forth between using a proxy, or not. The intent to use a proxy is indicated by supplying one more argument to the constructor of the database object of type `databaseUserType`. This argument is the IP-address of a Z++ Internet Server. The compiler does the rest of the work.

Exceptions

The following exceptions could be raised when declaring a database object, i.e. an instance of `databaseUserType`.

```
_EXCEPTION_DATABASE_UnsupportedDatabaseKind
_EXCEPTION_DATABASE_LibraryInitializationFailed
_EXCEPTION_DATABASE_ConnectionToServerFailed
```

The following exceptions can occur when carrying out an SQL statement. The first three exceptions are related to the select statement. The fetch exception could also occur when executing an explicit fetch-statement.

```
_EXCEPTION_DATABASE_SelectQueryFailed
_EXCEPTION_DATABASE_InsufficientMemoryForQueryResult
_EXCEPTION_DATABASE_FetchFailed

_EXCEPTION_DATABASE_InsertRequestFailed
_EXCEPTION_DATABASE_UpdateRequestFailed
_EXCEPTION_DATABASE_RemoveRequestFailed
```

Conclusion

The speed of execution is not a concern. In fact, if the database server responds fast, the select statement can receive hundreds of rows in a blink, over the Internet. This makes the other features of the model practically useful.

The model automates exchange of data between Z++ objects and database. The mechanism for this automation is quite intuitive. The mapping, in particular, is a simple list of fields of tables and their images as members of the catalyzer.

The model also automates the establishing of a session, and ending the session. A session begins with the declaration of an instance of `databaseUserType` and ends when the instance goes out of scope, or in case of dynamic objects when they are deleted. Thus, each Z++ thread can have multiple concurrent sessions with any number of databases.

An interesting fact is that SQL makes database systems platform independent, in a manner of speaking. Thus, a Z++ application can concurrently engage with multiple databases from different vendors.

Last but not the least, is the similarity of the statements to their SQL counterparts. Thus, instead of encoding and resorting to innovative programming tricks, an engineer uses familiar linguistic patterns. Finally, the use of proxy for communicating with a database server does not require code change beyond one intuitive argument to the constructor of the database object, the IP-address of the proxy.

Generally, models other than the one presented in this article are limited to a few specific platforms. In particular, the formulation illustrated in this article is a permanent pattern with clear semantics. In contrast, the use of other models, such as class libraries, requires continual maintenance resulting from library upgrades.

The database statements presented here are accompanied with a number of exceptions. The Z++ resumption capability provides a simple model for handling or correcting exceptional situations when dealing with complex database applications.