COCONUT Deliverable D13

Upgraded State of the Art Techniques Implemented as Modules

July 29, 2003

ILOG
TU Darmstadt
University of Coimbra
University of Vienna
University of Nantes
Catholic University of Louvain
Swiss Federal Institute of Technology

CHAPTER 3

An introduction to the Vienna Database Library

H. SCHICHL
University of Vienna, Department of Mathematics
A-1090 Wien, Austria
email: Hermann.Schichl@esi.ac.at

1. Introduction

The Vienna Database Library (VDBL) is a in memory database developed with generic programming in mind. It uses STL containers like `map` and `vector` to organize its internal structure.

Databases in the VDBL consist of tables, which are constructed from columns and rows as ordinary relational databases.

Columns can take arbitrary types, and their values need not be constant. Using function objects, called *methods*, the column values can change according to an *evaluation context*.

It is possible to construct *views* onto the tables of a database.

2. Database

A VDBL database consists of a number of tables which can be dynamically constructed, changed and destroyed. Every table (see Section 3) has a unique **name** (a `std::string`) and a unique **table id**, which is used for organizing the internal structure.

There is a general table interface defined in `class table`, which defines the minimal functionality needed for implementing a VDBL table. The structure is defined in such a way that `SQL` interfaces could be written, as well as tables which keep all their data in memory.

In addition to tables, the database knows of *users*. There are access control lists (at the moment not fully implemented) for restricting the access of users to the tables on a global and a column-wise base. The users are defined in the `class user`.

Users can construct **views** onto tables (see Section 6). These views can restrict a table to a subset of columns and/or rows. Also, additional rows can be defined for a view, and it is even possible to *join* various tables into one view. All views onto tables are constructed within a prespecified *context* (see Section 8). Using this mechanism, columns can change their value automatically according to the evaluation context. This is, e.g., useful in the COCONUT project for organizing points, where some of the properties change from work node to work node, like whether the point is feasible or not.

3. Tables

A VDBL table consists of a number of columns and rows. The definition of a table is always done by specifying its columns (see Section 4). The type of the columns value, which can be any `C++` type, is fixed upon creating the column. This can be done dynamically, like modifying and removing. Optionally, for each column a *default value* can be given (this default value may also change w.r.t. the evaluation context). All columns within a table have a **name** (a `std::string`) and a **column id**, which is used for organizing the column structure of the table internally. A column of a table can be accessed by specifying its name or, equivalently, its column id.

In addition to the column structure, which determines the outline of the table, the table's data is organized in **rows** (see Section 5). Every row has a **row id**, which is used for internal organization. Rows themselves consist of columns. When creating a new row, strict type checking is done between the row's column entries and the column type stored in the table. Column entries of a row can be left out, if a default value for the column is specified in the table definition.

It is possible to implement differently organized tables, as long as they are subclasses of the `class table`.

Implemented are two table subclasses:

Standard Table: A table which keeps all data in memory, internally organized as STL maps (`vdbl::standard_table`).

View Table: A table which is constructed from an arbitrary view using the view's internal structure to organize the data (`vdbl::view_table`).

4. Columns

VDBL columns are built in a very complicated way using three classes on top of each other, making it possible that arbitrary `C++` types can be stored in a column.

There are two main column classes implemented:

`typed_col`: This column holds constant values of arbitrary types. Their values are independent of the evaluation context (`class vdbl::typed_ol<_T>`).

`method_col`: A column of this type holds data, whose value is computed whenever it is retrieved and may depend on the evaluation context (`vdbl::method_col<_T>`). Instead of holding data, its contents are function objects (methods), which are subclasses of `class vdbl::method<_T>`. These function objects are used to calculate the column value.

Within a table different column types can be mixed within different rows and the default value, as long as their content types (strict run-time type checking) coincide.

5. Rows

The VDBL rows (`class vdbl::row`) are internally defined as STL maps of columns, organized with **column id** keys and column entries.

In principle, different types of rows could be defined, but at the moment only standard rows are implemented.

Every row contains a number of columns, whose values can be retrieved within an evaluation context (see Section 8). The column type within a row is arbitrary. If you want to make sure, that type checking is used, you have to change the rows through the table methods.

6. Views

A **view** (`class vdbl::view_base`) onto a table is table-like construct built from table structures. They may be restricted to a subset of the rows and/or columns of the table.

The most important properties of a view is that it is always created within a given context (see Section 8). The contents of the view can vary depending on this context. Two different views to the same table can at the same time show different data in the same column of the same row.

Two different classes of views have been implemented:

- **Standard View:** This view (of `class vdbl::view`) is constructed over **one** table, and it can only be restricted to subsets of rows and columns of this table.
- **Hierarchical View:** An hierarchical view (in `class vdbl::hierarchical_view`) looks onto a **stack of tables**, the top ones "overlaying" the lower ones. This makes it possible to have, e.g., a globally valid table on bottom and a stack of locally valid tables on top of them.

Some views can hold an internal **cache** of table entries, which are used for fast access, reducing the number of calls to function objects within columns.

7. View Database

A **view database** is a view onto a complete database, automatically constructing views (standard or hierarchical) for every table defined in the database. These views can be accessed under the same names as the defining tables, having a subset of their columns (also with identical names). The defining class is `vdbl::viewdbase`.

8. Contexts

Evaluation contexts are subclasses of `class vdbl::context`. They may hold arbitrary data and are keeping a `const vdbl::table *` to their associated table.

This context is passed to every function object along with the row the column belongs to for constructing the columns value. The contexts have no influence on `typed_col` columns, whose values don't change within different contexts.