GNATCOLL Documentation - Database packages

Release 26.0.0

AdaCore

Nov 10, 2026

CONTENTS

1 SQL: Database interface **1**
- 1.1 Database Abstraction Layers . . . 2
- 1.2 Database example . . . 3
- 1.3 Database schema . . . 3
- 1.4 The gnatcoll_db2ada tool . . . 6
  - 1.4.1 Default output of gnatcoll_db2ada . . . 9
  - 1.4.2 database introspection in Ada . . . 9
  - 1.4.3 Back to the library example... . . . 9
- 1.5 Connecting to the database . . . 10
- 1.6 Loading initial data in the database . . . 12
- 1.7 Writing queries . . . 14
- 1.8 Executing queries . . . 16
- 1.9 Prepared queries . . . 17
- 1.10 Getting results . . . 19
- 1.11 Creating your own SQL types . . . 20
- 1.12 Query logs . . . 21
- 1.13 Writing your own cursors . . . 22
- 1.14 The Object-Relational Mapping layer (ORM) . . . 23
  - 1.14.1 reverse relationships . . . 27
- 1.15 Modifying objects in the ORM . . . 28
- 1.16 Object factories in ORM . . . 29

2 Xref: Cross-referencing source code **33**

3 Xref: gnatinspect **37**

4 Indices and tables **41**

Index **43**

CHAPTER

ONE

SQL: DATABASE INTERFACE

A lot of applications need to provide **persistence** for their data (or a part of it). This means the data needs to be somehow saved on the disk, to be read and manipulated later, possibly after the application has been terminated and restarted. Although Ada provides various solutions for this (including the use of the streams as declared in the Ada Reference Manual), the common technique is through the use of relational database management systems (**RDBMS**. The term database is in fact overloaded in this context, and has come to mean different things:

- The software system that implements file and query management. This is generally provided by a third-party. The common abbreviation for these is **DBMS**. Queries are generally written in a language called **SQL**. One of the issues is that each DBMS tends to make minor changes to this language. Another issue is that the way to send these SQL commands to the DBMS is vendor-specific. GNATColl tries to abstract this communication through its own API. It currently supports **PostgreSQL** and **sqlite**, and makes it relatively easy to change between these two systems. For instance, development could be done using a local sqlite DBMS, and then deployed (after testing, of course!) on a PostgreSQL system.

  The code in GNATColl is such that adding support for a new DBMS should be relatively easy.

- A place where an application stores its data. The term **database** in this document refers to this meaning. In a relational database, this place is organized into tables, each of which contains a number of fields. A row in a table represents one object. The set of tables and their fields is called the **schema** of the database.

Traditionally, writing the SQL queries is done inline: special markers are inserted into your code to delimit sections that contain SQL code (as opposed to Ada code), and these are then preprocessed to generate actual code. This isn't the approach chosen in GNATColl: there are several drawbacks, in particular your code is no longer Ada and various tools will choke on it.

The other usual approach is to write the queries as strings, which are passed, via a DBMS-specific API, to the DBMS server. This approach is very fragile:

- The string might not contain **well-formed** SQL. This will unfortunately only be detected at run time when the DBMS complains.
- This is not **type safe**. You might be comparing a text field with an integer, for instance. In some cases, the DBMS will accept that (sqlite for instance), but in some other cases it won't (PostgreSQL). The result might then either raise an error, or return an empty list.
- There is a risk of **SQL injection**. Assuming the string is constructed dynamically (using Ada's *&* operator), it might be easy for a user to pass a string that breaks the query, and even destroys things in the database.
- As discussed previously, the SQL code might not be **portable** across DBMS. For instance, creating an automatically increment integer primary key in a table is DBMS specific.
- The string is fragile if the database **schema changes**. Finding whether a schema change impacts any of the queries requires looking at all the strings in your application.
- **performance** might be an issue. Whenever you execute a query, the DBMS will analyze it, decide how to execute it (for instance, whether it should traverse all the rows of a table, or whether it can do a faster lookup), and then

retrieve the results. The analysis pass is typically slow (relatively the overall execution time), and queries can in fact be **prepared** on the server: they are then analyzed only once, and it is possible to run them several times without paying the price of the analysis every time. Such a query can also be **parameterized**, in that some values can be derefed until the query is actually executed. All the above is made easy and portable in GNATColl, instead of requiring DBMS-specific techniques.

- This might require **large amount of code** to setup the query, bind the parameters, execute it, and traverse the list of results.

GNATColl attempts to solve all these issues. It also provides further performance improvements, for instance by keeping connections to the DBMS open and reusing them when possible. A paper was published at the Ada-Europe conference in 2008 which describes the various steps we went through in the design of this library.

1.1 Database Abstraction Layers

GNATColl organizes the API into several layers, each written on top of the previous one and providing more abstraction. All layers are compatible with each other and can be mixed inside a given application.

- **low-level binding**.

  This API is DBMS-specific, and is basically a mapping of the C API provided by the DBMS vendors into Ada. If you are porting C code, or working with an existing application, as a way to start using GNATColl before moving to higher levels of abstraction.

  The code is found in `gnatcoll-sql-sqlite-gnade.ads` and `gnatcoll-sql-postgres-gnade.ads`. The *gnade* part in the file names indicate that this code was initially inspired by the **GNADE** library that used to be available on the Internet. Part of the code might in fact come from that library.

  Using this API requires writing the SQL queries as strings, with all the disadvantages that were highlighted at the beginning of this chapter.

- **GNATCOLL.SQL** and **GNATCOLL.SQL.Exec**

  The first of these packages makes it possible to write type-safe queries strongly linked to the database schema (thus with a guarantee that the query is up-to-date with regards to the schema). To accomplish this, it also relies on code that is generated automatically from a description of your database schema, using the tool *gnatcoll_db2ada*. To simplify memory management, the queries are automatically referenced counted and freed when they are no longer needed.

  The second of these packages provides communication with the DBMS. It provides a vendor-neutral API. You can send your queries either as strings, or preferably as written with *GNATCOLL.SQL*. It also provides a simple way to prepare parameterized statements on the server for maximum efficiency, as well as the reuse of existing DBMS connections. It provides a simple API to retrieve and manipulate the results from a query.

- **GNATCOLL.SQL.ORM** and **GNATCOLL.SQL.Sessions**

  This is an Object-Relational Mapping (ORM).

  The first of these packages makes it possible to manipulate a database without writing SQL. Instead, you manipulate Ada objects (tagged types), whose primitive operations might transparently execute SQL queries. This API provides caching for maximum efficiency. It relies on code automatically generated by *gnatcoll_db2ada* from the schema of your database. The generated objects can then be extended in your own code if needed.

  The second package encapsulates DBMS connections into higher-level objects which provide their own caching and work best with the ORM objects. A session is automatically released to a pool when no longer needed and will be reused later on.

The following sections will ignore the lower layer, and concentrate on the other layers. They share a number of types and, again, are fully compatible with each other. You could connect to the database, and then write some queries using **GNATCOLL.SQL** and some using **GNATCOLL.SQL.ORM**.

1.2 Database example

This section describes an example that will be extended throughout this chapter. We will build an application that represents a library. Such a library contains various media (books and DVDs for instance), and customers. A customer can borrow multiple media at the same time, but a media is either at a customer's, or still in the library.

The GNATColl distribution includes an example directory which contains all the code and data for this example.

1.3 Database schema

As was mentioned earlier (*Database Abstraction Layers*), GNATColl relies on automatic code generation to provide a type safe interface to your database. This code is generated by an external tool called *gnatcoll_db2ada*. In some cases, this tool requires an installation of python (www.python.org) on your machine, since part of the code is written in that language.

This tool is able to output various kind of information, and will be fully described later (*The gnatcoll_db2ada tool*). However, the input is always the same: this is the schema of your database, that is the list of tables and fields that make up your database. There exist two ways to provide that information:

- From a running database

  If you pass the DBMS vendor (postgresql, sqlite,...) and the connection parameters to *gnatcoll_db2ada*, it is able to query the schema on its own. However, this should not be the preferred method: this is similar to reverse engineering assembly code into the original high-level code, and some semantic information will be missing. For instance, in SQL we have to create tables just to represent the many-to-many relationships. These extra tables are part of the implementation of the schema, but are just noise when it comes to the semantics of the schema. For this reason, it is better to use the second solution below:

- From a textual description

  Using the *-dbmodel* switch to *gnatcoll_db2ada*, you can pass a file that describes the schema. We do not use SQL as the syntax in this, because as explained above this is too low-level. This text file also provides additional capabilities that do not exist when reverse-engineering an existing database, for instance the ability to use name to represent reverse relationships for foreign keys (see below and the ORM).

  The most convenient editor for this file is Emacs, using the *org-mode* which provides convenient key shortcuts for editing the contents of ASCII tables. But any text editor will do, and you do not need to align the columns in this file.

  All lines starting with a hash sign ('#') will be ignored.

  This file is a collection of ASCII tables, each of which relates to one table or one SQL view in your database. The paragraphs start with a line containing:

```
table ::=
   '|' ('ABSTRACT')? ('TABLE'|'VIEW') ['(' supertable ')']
   '|' <name> '|' <name_row>
```

  "name" is the name of the table. The third pipe and third column are optional, and should be used to specify the name for the element represented by a single row. For instance, if the table is called "books", the third column could contain "book". This is used when generating objects for use with *GNATCOLL.SQL.ORM*.

  If the first line starts with the keyword *ABSTRACT*, then no instance of that table actually exists in the database. This is used in the context of table inheritance, so define shared fields only once among multiple tables.

  The keyword *TABLE* can be followed by the name of a table from which it inherits the fields. Currently, that supertable must be abstract, and the fields declared in that table are simply duplicated in the new table.

Following the declaration of the table, the file then describe their fields, each on a separate line. Each of these lines must start with a pipe character ("|"), and contain a number of pipe-separated fields. The order of the fields is always given by the following grammar:

```
fields ::=
   '|' <name> '|' <type>
   '|' ('PK'|''|'NULL'|'NOT NULL'|'INDEX'|'UNIQUE'|'NOCASE')
   '|' [default] '|' [doc] '|'
```

The type of the field is the SQL type ("INTEGER", "TEXT", "TIMESTAMP", "DATE", "DOUBLE PRECISION", "MONEY", "BOOLEAN", "TIME", "CHARACTER(1)"). Any maximal length can be specified for strings, not just 1 as in this example. The tool will automatically convert these to Ada when generating Ada code. A special type ("AUTOINCREMENT") is an integer that is automatically incremented according to available ids in the table. The exact type used will depend on the specific DBMS.

The property 'NOCASE' indicates that comparison should be case insensitive for this field.

If the field is a foreign key (that is a value that must correspond to a row in another table), you can use the special syntax for its type:

```
fk_type ::= 'FK' <table_name> [ '(' <reverse_name> ')' ]
```

As you can see, the type of the field is not specified explicitly, but will always be that of the foreign table's primary key. With this syntax, the foreign table must have a single field for its primary key. GNATColl does not force a specific order for the declaration of tables: if is valid to have a foreign key to a table that hasn't been declared yet. There is however a restriction if you use the model to create a sqlite database (through the *-createdb* switch of *gnatcoll_db2ada*): in this case, a reference to a table that hasn't been defined yet may not be not through a field marked as NOT NULL. This is a limitation of the sqlite backend itself. The solution in this case is to reorder the declaration of tables, or drop the NOT NULL constraint.

Another restriction is that a foreign key that is also a primary key must reference a table that has already been defined. You need to reorder the declaration of your tables to ensure this is the case.

"reverse_name" is the optional name that will be generated in the Ada code for the reverse relationship, in the context of *GNATCOLL.SQL.ORM*. If the "reverse_name" is empty (the parenthesis are shown), no reverse relationship is generated. If the parenthesis and the reverse_name are both omitted, a default name is generated based on the name of the field.

The third column in the fields definition indicates the constraints of the type. Multiple keywords can be used if they are separated by commas. Thus, "NOT NULL, INDEX" indicates a column that must be set by the user, and for which an index is created to speed up look ups.

- A primary key ("PK")
- The value must be defined ("NOT NULL")
- The value can be left undefined ("NULL")
- A unique constraint and index ("UNIQUE")
- An index should be created for that column ("INDEX") to speed up the lookups.
- The automatic index created for a Foreign Key should not be created ("NOINDEX"). Every time a field references another table, GNATColl will by default create an index for it, so that the ORM can more efficiently do a reverse query (from the target table's row find all the rows in the current table that reference that target row). This will in general provide more efficiency, but in some cases you never intend to do the reverse query and thus can spare the extra index.

The fourth column gives the default value for the field, and is given in SQL syntax. Strings must be quoted with single quotes.

The fifth column contains documentation for the field (if any). This documentation will be included in the generated code, so that IDEs can provide useful tooltips when navigating your application's code.

After all the fields have been defined, you can specify extract constraints on the table. In particular, if you have a foreign key to a table that uses a tuple as its primary key, you can define that foreign key on a new line, as:

```
FK ::= '|' "FK:" '|' <table> '|' <field_names>*
   '|' <field_names>* '|'
```

For instance:

```
| TABLE | tableA |
| FK: | tableB | fieldA1, fieldA2 | fieldB1, fieldB2 |
```

It is also possible to create multi-column indexes, as in the following example. In this case, the third column contains the name of the index to create. If left blank, a default name will be computed by GNATColl:

```
| TABLE | tableA |
| INDEX: | field1,field2,field3 | name |
```

The same way the unique multi-column constraint and index can be created. The name is optional.

TABLE | tableA |
UNIQUE: | field1,field2,field3 | name |

Going back to the example we described earlier (*Database example*), let's describe the tables that are involved.

The first table contains the customers. Here is its definition:

```
| TABLE | customers     | customer        || The customer for the library |
| id    | AUTOINCREMENT | PK              || Auto-generated id            |
| first | TEXT          | NOT NULL        || Customer's first name        |
| last  | TEXT          | NOT NULL, INDEX || Customer's last name         |
```

We highly recommend to set a primary key on all tables. This is a field whose value is unique in the table, and thus that can act as an identifier for a specific row in the table (in this case for a specific customer). We recommend using integers for these ids for efficiency reasons. It is possible that the primary key will be made of several fields, in which case they should all have the "PK" constraint in the third column.

A table with no primary key is still usable. The difference is in the code generated for the ORM (*The Object-Relational Mapping layer (ORM)*), since the *Delete* operation for this table will raise a *Program_Error* instead of doing the actual deletion (that's because there is no guaranteed unique identifier for the element, so the ORM does not know which one to delete – we do not depend on having unique internal ids on the table, like some DBMS have). Likewise, the elements extracted from such a primary key-less table will not be cached locally in the session, and cannot be updated (only new elements can be created in the table).

As we mentioned, the library contains two types of media, books and DVDs. Each of those has a title, an author. However, a book also has a number of pages and a DVD has a region where it can be viewed. There are various ways to represent this in a database. For illustration purposes, we will use table inheritance here: we will declare one abstract table (media) which contains the common fields, and two tables to represent the types of media.

As we mentioned, a media can be borrowed by at most one customer, but a customer can have multiple media at any point in time. This is called a **one-to-many** relationship. In SQL, this is in general described through the use of a foreign key that goes from the table on the "many" side. In this example, we therefore have a foreign key from media to customers. We also provide a name for the reverse relationship, which will become clearer when we describe the ORM interface.

Here are the declarations:

```
| ABSTRACT TABLE | media               | media || The contents of the library |
| id             | AUTOINCREMENT       | PK    || Auto-generated id           |
| title          | TEXT                |       || The title of the media      |
| author         | TEXT                |       || The author                  |
| published      | DATE                |       || Publication date            |
| borrowed_by    | FK customers(items) | NULL  || Who borrowed the media      |

| TABLE (media) | books   | book |     | The books in the library |
| pages         | INTEGER |      | 100 |                          |

| TABLE (media) | dvds    | dvd |   | The dvds in the library |
| region        | INTEGER |     | 1 |                         |
```

For this example, all this description is put in a file called `dbschema.txt`.

1.4 The gnatcoll_db2ada tool

As stated in the introduction, one of the goals of this library is to make sure the application's code follows changes in the schema of your database.

To reach this goal, an external tool, `gnatcoll_db2ada` is provided with GNATColl, and should be spawned as the first step of the build process, or at least whenever the database schema changes. It generates an Ada package (*Database* by default) which reflects the current schema of the database.

This tool supports a number of command line parameters (the complete list of which is available through the `-h` switch). The most important of those switches are:

-dbhost host*, *-dbname name*, *-dbuser user*, *-dbpasswd passwd*, *-dbtype type
: These parameters specify the connection parameters for the database. To find out the schema, `gnatcoll_db2ada` can connect to an existing database (*Database schema*). The user does not need to have write permission on the database, since all queries are read-only.

-dbmodel file
: This parameter can replace the above *-dbname*,... It specifies the name of a text file that contains the description of the database, therefore avoiding the need for already having a database up-and-running to generate the Ada interface.

  The format of this text file was described in the previous section.

  This switch is not compatible with *-enum* and *-vars* that really need an access to the database.

-api PKG
: This is the default behavior if you do not specify *-text* or *-createdb*. This will generate several files (`PKG.ads`, `PKG.adb` and `PKG_names.ads`, where PKG is the argument given on the command line). These package represent your database schema, that is the list of all tables and their fields, as typed values. This is the building block for using *GNATCOLL.SQL* and write type-safe queries.

-api-enums PKG
: This is very similar to *-api*, except it will only extract values from an existing database as per the *-enum* and *-var* switches. The generated package does not include the description of the database schema. The goal is that the values are extracted once from an existing database, and then *-api* can be used to dump the schema from a textual description of the database.

-adacreate
: This should be used in combination with *-api*. In addition to the usual output of *-api*, it will also generate an Ada subprogram called *Create_Database* that can be used to recreate the database and its initial data (if *-load*

was specified) from your application, without requiring access to the external files that define the schema and the initial data.

-enum table,id,name,prefix,base

This parameter can be repeated several times if needed. It identifies one of the special tables of the database that acts as an enumeration type. It is indeed often the case that one or more tables in the database have a role similar to Ada's enumeration types, i.e. contains a list of values for information like the list of possible priorities, a list of countries,... Such lists are only manipulated by the maintainer of the database, not interactively, and some of their values have impact on the application's code (for instance, if a ticket has an urgent priority, we need to send a reminder every day – but the application needs to know what an urgent priority is). In such a case, it is convenient to generate these values as constants in the generated package. The output will be similar to:

```
subtype Priority_Id is Integer;
Priority_High   : constant Priority_Id := 3;
Priority_Medium : constant Priority_Id := 2;
Priority_Low    : constant Priority_Id := 1;
Priority_High_Internal : constant Priority_Id := 4;
```

This code would be extracted from a database table called, for instance, *ticket_priorities*, which contains the following:

```
table ticket_priorities:
name                | priority      | category
high                | 3             | customer
medium      | 2             | customer
low         | 1             | customer
high_internal       | 4             | internal
```

To generate the above Ada code, you need to pass the following parameter to `gnatcoll_db2ada`:

```
-enum ticket_priorities,Priority,Name,Priority,Integer
```

First word in the parameter is the table name where the data to generate constants is stored. Second word is the field name in the table where the Ada constant value is stored. The third word is the field where the last part the Ada constant name is stored. The forth word is the prefix to add in front of the third word field value to generate the Ada constant's name. The last optional parameter should be either *Integer* (default) or *String*, which influences the way how the Ada constant value is going to be generated (surrounded or not by quotes).

-enum-image

If specified in addition to the *-enum* switch, then a function is generated for each *Integer*-valued enum that converts numeric values to the corresponding name as a string.

This function is generated as an Ada 2012 expression-function such as:

```
function Image_Priority_Id (X : Priority_Id) return String is
  (case X is
      when 3      => "High",
      when 2      => "Medium",
      when 1      => "Low",
      when 4      => "High_Internal",
      when others => raise Constraint_Error
                       with "invalid Priority_Id " & X'Img);
```

-var name,table,field,criteria,comment

This is similar to the *-enum* switch, but extracts a single value from the database. Although applications should try and depend as little as possible on such specific values, it is sometimes unavoidable.

For instance, if we have a table in the table with the following contents:

```
table staff
staff_id    | login
0           | unassigned
1           | user1
```

We could extract the id that helps detect unassigned tickets with the following command line:

```
-var no_assign_id,staff,staff_id,"login='unassigned'","help"
```

which generates:

```
No_Assigne_Id : constant := 0;
--  help
```

The application should use this constant rather than some hard-coded string *"unassigned"* or a named constant with the same value. The reason is that presumably the login will be made visible somewhere to the user, and we could decide to change it (or translate it to another language). In such a case, the application would break. On the other hand, using the constant *0* which we just extracted will remain valid, whatever the actual text we display for the user.

-orm PKG

This will generate two files (`PKG.ads` and `PKG.adb`) that support *GNATCOLL.SQL.ORM* to write queries without writing actual SQL. This is often used in conjunction with `-api`, as in:

```
gnatcoll_db2ada -api Database -orm ORM -dbmodel dbschema.txt
```

To use this switch, you need to have a version of python installed on your development machine, since the code generation is currently implemented in python. The generated code can then be compiled on any machine, so it is enough to generate the code once and then possibly check it in your version control system.

-ormtables LIST

Restrict the output of *-orm* to a subset of the tables. List is a comma-separated of table names.

-dot

When this switch is specified, *gnatcoll_db2ada* generates a file called `schema.dot` in the current directory. This file can be processed by the *dot* utility found in the *graphviz* suite, to produce a graphical representation of your database schema. Each table is represented as a rectangle showing the list of all attributes, and the foreign keys between the tables are represented as links.

To produce this output, you need a python installation on your machine. If you also have *dot* installed, the file is processed automatically to generate a postscript document `schema.ps`.

-text

Instead of creating Ada files to represent the database schema, this switch will ask *gnatcoll_db2ada* to dump the schema as text. This is in a form hopefully easy to parse automatically, in case you have tools that need the schema information from your database in a DBMS-independent manner. This is the same format used for *-dbmodel*, so the switch *-text* can also be used to bootstrap the development if you already have an existing database.

-createdb

Instead of the usual default output, *gnatcoll_db2ada* will output a set of SQL commands that can be used to recreate the set of all tables in your schema. This does not create the database itself (which might require special rights depending on your DBMS), only the tables.

In most cases, this creation needs to be done by a system administrator with the appropriate rights, and thus will be done as part of the deployment of your application, not the application itself. This is particularly true for client-server databases like *PostgreSQL*.

But in some simpler cases where the database is only manipulated by your application, and potentially only needs to exist while your application is running (often the case for *sqlite*), your application could be responsible for creating the database.

1.4.1 Default output of gnatcoll_db2ada

From the command line arguments, *gnatcoll_db2ada* will generate an Ada package, which contains one type per table in the database. Each of these types has a similar structure. The implementation details are not shown here, since they are mostly irrelevant and might change. Currently, a lot of this code are types with discriminants. The latter are *access-to-string*, to avoid duplicating strings in memory and allocating and freeing memory for these. This provides better performances:

```
package Database is
   type T_Ticket_Priorities (...) is new SQL_Table (...) with record
       Priority : SQL_Field_Integer;
       Name     : SQL_Field_Text;
   end record;

   overriding function FK (Self : T_Ticket_Priorities; Foreign : SQL_Table'Class)
      return SQL_Criteria;

   Ticket_Priorities : constant T_Ticket_Priorities (...);
end Database;
```

It provides a default instance of that type, which can be used to write queries (see the next section). This type overrides one primitive operation which is used to compute the foreign keys between that table and any other table in the database (*Writing queries*).

Note that the fields which are generated for the table (our example reuses the previously seen table *ticket_priorities*) are typed, which as we will see provides a simple additional type safety for our SQL queries.

1.4.2 database introspection in Ada

As described above, the *-createdb* switch makes it possible to create a database (or at least its schema). This operation can also be performed directly from your Ada code by using the services provided in the *GNATCOLL.SQL.Inspect* package. In particular, there are services for reading the schema of a database either from a file or from a live database, just as *gnatcoll_db2ada* does.

This results in a structure in memory that you can use to find out which are the tables, what are their fields, their primary keys,...

It is also possible to dump this schema to a text file (with the same format as expected by *-dbmodel*), or more interestingly to output the SQL statements that are needed to create the tables in a database. In the case of Sqlite, creating a table will also create the database file if it doesn't exist yet, so no special rights are needed.

This input/output mechanism is implemented through an abstract *Schema_IO* tagged type, with various concrete implementations (either *File_Schema_IO* to read or write from/to a file, or *DB_Schema_IO* to read or write from/to a database).

See the specs for more detail on these subprograms.

1.4.3 Back to the library example...

In the previous section, we have described our database schema in a text file. We will now perform two operations:

- Create an empty database

  This should of course only be done once, not every time you run your application:

```
gnatcolldbada -dbtype=sqlite -dbname=library.db -dbmodel=dbschema.txt -createdb
```

In the case of this example, the sql commands that are executed for sqlite are:

```
CREATE TABLE books (
   pages Integer DEFAULT '100',
   id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   title Text,
   author Text,
   published Date,
   borrowed_by Integer);
CREATE TABLE customers (
   id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   first Text NOT NULL,
   last Text NOT NULL);
CREATE TABLE dvds (
   region Integer DEFAULT '1',
   id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   title Text,
   author Text,
   published Date,
   borrowed_by Integer);
CREATE INDEX "customers_last" ON "customers" ("last");
```

- Generate the Ada code

  The details of the code will be described later. For now, our application will not use the ORM, so we do not generate code for it:

```
gnatcoll_db2ada -api=Database -dbmodel=dbschema.txt
```

1.5 Connecting to the database

This library abstracts the specifics of the various database engines it supports. Ideally, code written for one database could be ported almost transparently to another engine. This is not completely doable in practice, since each system has its own SQL specifics, and unless you are writing things very carefully, the interpretation of your queries might be different from one system to the next.

However, the Ada code should remain untouched if you change the engine. Various engines are supported out of the box (PostgreSQL and Sqlite), although new ones can be added by overriding the appropriate SQL type (*Database_Connection*). When you compile GNATColl, the build scripts will try and detect what systems are installed on your machine, and only build support for those. It is possible, if no database was installed on your machine at that time, that the database interface API is available (and your application compiles), but no connection can be done to database at run time.

To connect to a DBMS, you need to specify the various connection parameters. This is done via a *GNATCOLL.SQL.Exec.Database_Description* object. The creation of this object depends on the specific DBMS you are connecting to (and this is the only part of your code that needs to know about the specific system). The packages *GNATCOLL.SQL.Postgres* and *GNATCOLL.SQL.Sqlite* contain a *Setup* function, whose parameters depend on the DBMS. They provide full documentation for their parameters. Let's take a simple example from sqlite:

```
with GNATCOLL.SQL.Sqlite;   -- or Postgres
declare
   DB_Descr : GNATCOLL.SQL.Exec.Database_Description;
```

(continues on next page)

(continued from previous page)

```
begin
   DB_Descr := GNATCOLL.SQL.Sqlite.Setup ("dbname.db");
end
```

At this point, no connection to the DBMS has been done, and no information was exchanged.

To communicate with the database, however, we need to create another object, a **GNATCOLL.SQL.Exec.Database_Connection**. Your application can create any number of these. Typically, one would create one such connection per task in the application, although other strategies are possible (like a pool of reusable connections, where a task might be using two connections and another task none at any point in time).

If you do not plan on using the ORM interface from **GNATCOLL.SQL.ORM**, GNATColl provides a simple way to create a task-specific connection. While in this task, the same connection will always be returned (thus you do not have to pass it around in parameter, although the latter might be more efficient):

```
declare
   DB : GNATCOLL.SQL.Exec.Database_Connection;
begin
   DB := GNATCOLL.SQL.Exec.Get_Task_Connection
         (Description  => DB_Descr);
end;
```

If your application is not multi-tasking, or you wish to implement your own strategy for a connection pool, you can also use the following code (using Ada 2005 dotted notation when calling the primitive operation). This code will always create a new connection, not reuse an existing one, as opposed to the code above:

```
declare
   DB : GNATCOLL.SQL.Exec.Database_Connection;
begin
   DB := DB_Descr.Build_Connection;
end;
```

A note on concurrency: if you implement your own pool, you might sometimes end up with dead locks when using sqlite. If a task uses two or more connections to sqlite, and you setup GNATCOLL to create SQL transactions even for *SELECT* statements (see *GNATCOLL.SQL.Sqlite.Always_Use_Transactions*), the following scenario will result in a deadlock:

```
DB1 := ... new connection to sqlite
   ... execute a SELECT through DB1. The latter then holds a shared
   ... lock, preventing other connections from writing (but not from
   ... reading).
DB2 := ... another connection in the same thread
   ... execute an INSERT through DB2. This tries to get a lock, which
   ... will fail while DB1 holds the shared lock. Since these are in
   ... the same thread, this will deadlock.
```

By default, GNATCOLL will not create SQL transactions for select statements to avoid this case, which occurs frequently in code.

If you wish to reuse an existing connection later on, you must reset it. This terminates any on-going SQL transaction, and resets various internal fields that describe the state of the connection:

```
Reset_Connection (DB);
```

In all three cases, the resulting database connection needs to be freed when you no longer needed (which might be when your program terminates if you are using pools) to avoid memory leaks. Nothing critical will appear if you do not close, though, because the transactions to the DBMS server are saved every time you call *Commit* in any case. So the code would end with:

```
Free (DB);  --  for all connections you have opened
Free (DB_Descr);
```

At this point, there still hasn't been any connection to the DBMS. This will be done the first time a query is executed. If for some reason the connection to the DBMS server is lost, GNATColl will automatically attempt to reconnect a number of times before it gives up. This might break if there was an ongoing SQL transaction, but simplifies your code since you do not have to handle reconnection when there was a network failure, for instance.

As we saw before, the database interface can be used in multi-tasking applications. In such a case, it is recommended that each thread has its own connection to the database, since that is more efficient and you do not have to handle locking. However, this assumes that the database server itself is thread safe, which most often is the case, but not for *sqlite* for instance. In such a case, you can only connect one per application to the database, and you will have to manage a queue of queries somehow.

If you want to use **GNATCOLL.SQL.Sessions** along with the Object-Relational Mapping API, you will need to initialize the connection pool with the **Database_Description**, but the session will then take care automatically of creating the **Database_Connection**. See later sections for more details.

1.6 Loading initial data in the database

We have now created an empty database. To make the queries we will write later more interesting, we are going to load initial data.

There are various ways to do it:

- Manually or with an external tool

  One can connect to the database with an external tool (a web interface when the DBMS provides one for instance), or via a command line tool (*psql* for PostgreSQL or *sqlite3* for Sqlite), and start inserting data manually. This shows one of the nice aspects of using a standard DBMS for your application: you can alter the database (for instance to do minor fixes in the data) with a lot of external tools that were developed specifically for that purpose and that provide a nice interface. However, this is also tedious and error prone, and can't be repeat easily every time we recreate the database (for instance before running automatic tests).

- Using *GNATCOLL.SQL.EXEC*

  As we will describe later, GNATColl contains all the required machinery for altering the contents of the database and creating new objects. Using *GNATCOLL.SQL.ORM* this can also be done at a high-level and completely hide SQL.

- Loading a data file

  A lot of frameworks call such a file that contains initial data a "fixture". We will use this technique as an example. At the Ada level, this is a simple call to *GNATCOLL.SQL.Inspect.Load_Data*. The package contains a lot more than just this subprogram (*The gnatcoll_db2ada tool*):

  ```
  declare
     File : GNATCOLL.VFS.Virtual_File := Create ("fixture.txt");
     DB : Database_Connection;  --  created earlier
  begin
     GNATCOLL.SQL.Inspect.Load_Data (DB, File);
     DB.Commit;
  end;
  ```

The format of this file is described just below.

As we mentioned, GNATColl can load data from a file. The format of this file is similar to the one that describes the database schema. It is a set of ASCII tables, each of which describes the data that should go in a table (it is valid to duplicate tables). Each block starts with two lines: The first one has two mandatory columns, the first of which contains the text "TABLE", and the second contains the name of the table you want to fill. The second line should contain as many columns as there are fields you want to set. Not all the fields of the table need to have a corresponding column if you want to set their contents to NULL (provided, of course, that your schema allows it). For instance, we could add data for our library example as such:

```
| TABLE | customers |        |
|    id | first     | last   |
|-------+-----------+--------/
|     1 | John      | Smith  |
|     2 | Alain     | Dupont |

| TABLE      | books   |       |            |             |
| title      | author  | pages |  published | borrowed_by |
|------------+---------+-------+------------+-------------/
| Art of War | Sun Tzu |    90 | 01-01-2000 |           1 |
| Ada RM     | WRG     |   250 | 01-07-2005 |             |
```

A few comments on the above: the *id* for *books* is not specified, although the column is the primary key and therefore cannot be NULL. In fact, since the type of the *id* was set to AUTOINCREMENT, GNATColl will automatically assign valid values. We did not use this approach for the id of *customers*, because we need to know this id to set the *borrowed_by* field in the *books* table.

There is another approach to setting the *borrowed_by* field, which is to give the value of another field of the *customers* table. This of course only work if you know this value is unique, but that will often be the case in your initial fixtures. Here is an example:

```
| TABLE        | dvds      |        |                     |
| title        | author    | region | borrowed_by(&last) |
|--------------+-----------+--------+--------------------/
| The Birds    | Hitchcock |      1 | &Smith              |
| The Dictator | Chaplin   |      3 | &Dupont             |
```

Here, the title of the column indicates that any value in this column might be a reference to the *customers.last* value. Values which start with an ampersand ("&") will therefore be looked up in *customers.last*, and the *id* of the corresponding customer will be inserted in the *dvds* table. It would still be valid to use directly customer ids instead of references, this is just an extra flexibility that the references give you to make your fixtures more readable.

However, if we are using such references we need to provide the database schema to *Load_Data* so that it can write the proper queries. This is done by using other services of the *GNATCOLL.SQL.Inspect* package.

The code for our example would be:

```
Load_Data
   (DB, Create ("fixture.txt"),
    New_Schema_IO (Create ("dbschema.txt")).Read_Schema);
```

1.7 Writing queries

The second part of the database support in GNATColl is a set of Ada subprograms which help write SQL queries. Traditional ways to write such queries have been through embedded SQL (which requires a pre-processing phase and complicate the editing of source files in Ada-aware editors), or through simple strings that are passed as is to the server. In the latter case, the compiler can not do any verification on the string, and errors such a missing parenthesis or misspelled table or field names will not be detected until the code executes the query.

GNATColl tries to make sure that code that compiles contains syntactically correct SQL queries and only reference existing tables and fields. This of course does not ensure that the query is semantically correct, but helps detect trivial errors as early as possible.

Such queries are thus written via calls to Ada subprograms, as in the following example:

```
with GNATCOLL.SQL;  use GNATCOLL.SQL;
with Database; use Database;
declare
   Q : SQL_Query;
begin
   Q := SQL_Select
     (Fields => Max (Ticket_Priorities.Priority)
         & Ticket_Priorities.Category,
      From   => Ticket_Priorities,
      Where  => Ticket_Priorities.Name /= "low",
      Group_By => Ticket_Priorities.Category);
end;
```

The above example will return, for each type of priority (internal or customer) the highest possible value. The interest of this query is left to the user...

This is very similar to an actual SQL query. Field and table names come from the package that was automatically generated by the *gnatcoll_db2ada* tool, and therefore we know that our query is only referencing existing fields. The syntactic correctness is ensured by standard Ada rules. The *SQL_Select* accepts several parameters corresponding to the usual SQL attributes like *GROUP BY*, *HAVING*, *ORDER BY* and *LIMIT*.

The *From* parameter could be a list of tables if we need to join them in some ways. Such a list is created with the overridden *"&"* operator, just as for fields which you can see in the above example. GNATColl also provides a *Left_Join* function to join two tables when the second might have no matching field (see the SQL documentation).

Similar functions exist for *SQL_Insert*, *SQL_Update* and *SQL_Delete*. Each of those is extensively documented in the `gnatcoll-sql.ads` file.

It is worth noting that we do not have to write the query all at once. In fact, we could build it depending on some other criteria. For instance, imagine we have a procedure that does the query above, and omits the priority specified as a parameter, or shows all priorities if the empty string is passed. Such a procedure could be written as:

```
procedure List_Priorities (Omit : String := "") is
  Q : SQL_Query;
  C : SQL_Criteria := No_Criteria;
begin
  if Omit /= "" then
     C := Ticket_Priorities.Name /= Omit;
  end if;
  Q := SQL_Select
    (Fields => ..., -- as before
     Where  => C);
end;
```

With such a code, it becomes easier to create queries on the fly than it would be with directly writing strings.

The above call has not sent anything to the database yet, only created a data structure in memory (more precisely a tree). In fact, we could be somewhat lazy when writing the query and rely on auto-completion, as in the following example:

```
Q := SQL_Select
 (Fields => Max (Ticket_Priorities.Priority)
     & Ticket_Priorities.Category,
  Where  => Ticket_Priorities.Name /= "low");

Auto_Complete (Q);
```

This query is exactly the same as before. However, we did not have to specify the list of tables (which GNATColl can compute on its own by looking at all the fields referenced in the query), nor the list of fields in the *GROUP BY* clause, which once again can be computed automatically by looking at those fields that are not used in a SQL aggregate function. This auto-completion helps the maintenance of those queries.

There is another case where GNATColl makes it somewhat easier to write the queries, and that is to handle joins between tables. If your schema was build with foreign keys, GNATColl can take advantage of those.

Going back to our library example, let's assume we want to find out all the books that were borrowed by the user "Smith". We need to involve two tables (*Books* and *Customers*), and provide a join between them so that the DBMS knows how to associate the rows from one with the rows from the other. Here is a first example for such a query:

```
Q := SQL_Select
   (Fields => Books.Title & Books.Pages,
    From   => Books & Customers,
    Where  => Books.Borrowed_By = Customers.Id
       and Customers.Last = "Smith");
```

In fact, we could also use auto-completion, and let GNATColl find out the involved tables on its own. We thus write the simpler:

```
Q := SQL_Select
   (Fields => Books.Title & Books.Pages,
    Where  => Books.Borrowed_By = Customers.Id
       and Customers.Last = "Smith");
```

There is one more things we can do to simplify the query and make it more solid if the schema of the database changes. For instance, when a table has a primary key made up of several fields, we need to make sure we always have an "=" statement in the WHERE clause for all these fields between the two tables. In our example above, we could at some point modify the schema so that the primary key for *customers* is multiple (this is unlikely in this example of course). To avoid this potential problems and make the query somewhat easier to read, we can take advantage of the *FK* subprograms generated by *gnatcoll_db2ada*. Using the Ada05 dotted notation for the call, we can thus write:

```
Q := SQL_Select
   (Fields => Books.Title & Books.Pages,
    Where  => Books.FK (Customers)
       and Customers.Last = "Smith");
```

Regarding memory management, there is no need for explicitly freeing memory in the above code. GNATColl will automatically do this when the query is no longer needed.

1.8 Executing queries

Once we have our query in memory, we need to pass it on to the database server itself, and retrieve the results.

Executing is done through the *GNATCOLL.SQL.Exec* package, as in the following example:

```
declare
   R : Forward_Cursor;
begin
   R.Fetch (Connection => DB, Query => Q);
end;
```

This reuses the connection we have established previously (*DB*) (although now we are indeed connecting to the DBMS for the first time) and sends it the query. The result of that query is then stored in *R*, to be used later.

Some SQL commands execute code on the DBMS, but do not return a result. In this case, you can use *Execute* instead of *Fetch*. This is the case when you execute an *INSERT* or *UPDATE* statement for instance. Using *Execute* avoids the need to declare the local variable *R*.

If for some reason the connection to the database is no longer valid (a transient network problem for instance), GNATColl will attempt to reconnect and re-execute your query transparently, so that your application does not need to handle this case.

We'll describe later (*Getting results*) how to analyze the result of the query.

Some versions of *Fetch* have an extra parameter *Use_Cache*, set to *False* by default. If this parameter is true, and the exact same query has already been executed before, its result will be reused without even contacting the database server. The cache is automatically invalidated every hour in any case. This cache is mostly useful for tables that act like enumeration types, as we have seen before when discussing the *-enum* parameter to `gnatcoll_db2ada`. In this case, the contents of the table changes very rarely, and the cache can provide important speedups, whether the server is local or distant. However, we recommend that you do actual measurements to know whether this is indeed beneficial for you. You can always invalidate the current cache with a call to *Invalidate_Cache* to force the query to be done on the database server.

If your query produces an error (whether it is invalid, or any other reason), a flag is toggled in the *Connection* parameter, which you can query through the *Success* subprogram. As a result, a possible continuation of the above code is:

```
if Success (DB) then
   ...
else
   ...  --  an error occurred
end if
```

GNATColl also tries to be helpful in the way it handles SQL transactions. Such transactions are a way to execute your query in a sandbox, i.e. without affecting the database itself until you decide to *COMMIT* the query. Should you decide to abort it (or *ROLLBACK* as they say for SQL), then it is just as if nothing happened. As a result, it is in general recommended to do all your changes to the database from within a transaction. If one of the queries fail because of invalid parameters, you just rollback and report the error to the user. The database is still left in a consistent state. As an additional benefit, executing within a transaction is sometimes faster, as is the case for PostgreSQL for instance.

To help with this, GNATColl will automatically start a transaction the first time you edit the database. It is then your responsibility to either commit or rollback the transaction when you are done modifying. A lot of database engines (among which PostgreSQL) will not accept any further change to the database if one command in the transaction has failed. To take advantage of this, GNATColl will therefore not even send the command to the server if it is in a failure state.

Here is code sample that modifies the database:

```
Execute (DB, SQL_Insert (...));
--  Executed in the same transaction

Commit_Or_Rollback (DB);
--  Commit if both insertion succeeded, rollback otherwise
--  You can still check Success(DB) afterward if needed
```

1.9 Prepared queries

The previous section showed how to execute queries and statements. But these were in fact relatively inefficient.

With most DBMS servers, it is possible to compile the query once on the server, and then reuse that prepared query to significantly speed up later searches when you reuse that prepared statement.

It is of course pretty rare to run exactly the same query or statement multiple times with the same values. For instance, the following query would not give much benefit if it was prepared, since you are unlikely to reuse it exactly as is later on:

```
SELECT * FROM data WHERE id=1
```

SQL (and GNATColl) provide a way to parameterize queries. Instead of hard-coding the value *1* in the example above, you would in fact use a special character (unfortunately specific to the DBMS you are interfacing to) to indicate that the value will be provided when the query is actually executed. For instance, *sqlite* would use:

```
SELECT * FROM data WHERE id=?
```

You can write such a query in a DBMS-agnostic way by using GNATColl. Assuming you have automatically generated `database.ads` by using *gnatcoll_db2ada*, here is the corresponding Ada code:

```
with Database;  use Database;

Q : constant SQL_Query :=
    SQL_Select
       (Fields => Data.Id & Data.Name
        From   => Data,
        Where  => Data.Id = Integer_Param (1));
```

GNATColl provides a number of functions (one per type of field) to indicate that the value is currently unbound. *Integer_Param*, *Text_Param*, *Boolean_Param*,... All take a single argument, which is the index of the corresponding parameter. A query might need several parameters, and each should have a different index. On the other hand, the same parameter could be used in several places in the query.

Although the query above could be executed as is by providing the values for the parameters, it is more efficient, as we mentioned at the beginning, to compile it on the server. In theory, this preparation is done within the context of a database connection (thus cannot be done for a global variable, where we do not have connections yet, and where the query might be executed by any connection later on).

GNATColl will let you indicate that the query should be prepared. This basically sets up some internal data, but does not immediately compile it on the server. The first time the query is executed in a given connection, though, it will first be compiled. The result of this compilation will be reused for that connection from then on. If you are using a second connection, it will do its own compilation of the query.

So in our example we would add the following global variable:

```
P : constant Prepared_Statement :=
   Prepare (Q, On_Server => True);
```

Two comments about this code:

- You do not have to use global variables. You can prepare the statement locally in a subprogram. A *Prepared_Statement* is a reference counted type, that will automatically free the memory on the server when it goes out of scope.
- Here, we prepared the statement on the server. If we had specified *On_Server => False*, we would still have sped things up, since Q would be converted to a string that can be sent to the DBMS, and from then on reused that string (note that this conversion is specific to each DBMS, since they don't always represent things the same way, in particular parameters, as we have seen above). Thus every time you use P you save the time of converting from the GNATColl tree representation of the query to a string for the DBMS.

Now that we have a prepared statement, we can simply execute it. If the statement does not require parameters, the usual *Fetch* and *Execute* subprograms have versions that work exactly the same with prepared statements. They also accept a *Params* parameter that contains the parameter to pass to the server. A number of "+" operators are provided to create those parameters:

```
declare
   F : Forward_Cursor;
begin
   F.Fetch (DB, P, Params => (1 => +2));
   F.Fetch (DB, P, Params => (1 => +3));
end;
```

Note that for string parameters, the "+" operator takes an access to a string. This is for efficiency, to avoid allocating memory and copying the string, and is safe because the parameters are only needed while *Fetch* executes (even for a *Forward_Cursor*.

Back to our library example. We showed earlier how to write a query that retrieves the books borrowed by customer "Smith". We will now make this query more general: given a customer name, return all the books he has borrowed. Since we expect to use this often, we will prepare it on the server (in real life, this query is of little interest since the customer name is not unique, we would instead use a query that takes the id of the customer). In general we would create a global variable with:

```
Borrowed : constant Prepared_Statement := Prepare
  (SQL_Select
     (Fields => Books.Title & Books.Pages,
      Where  => Books.FK (Customers)
        and Customers.Last = Text_Param (1));
   Auto_Complete => True,
   On_Server => True);
```

Then when we need to execute this query, we would do:

```
declare
   Name : aliased String := "Smith";
begin
   R.Fetch (DB, Borrowed, Params => (1 => +Smith'Access));
end;
```

There is one last property on *Prepared_Statement`s: when you prepare them, you can pass a `Use_Cache => True* parameter. When this is used, the result of the query will be cached by GNATColl, and reuse when the query is executed again later. This is the fastest way to get the query, but should be used with care, since it will not detect

changes in the database. The local cache is automatically invalidated every hour, so the query will be performed again at most one hour later. Local caching is disabled when you execute a query with parameters. In this case, prepare the query on the server which will still be reasonably fast.

Finally, here are some examples of timings. The exact timing are irrelevant, but it is interesting to look at the different between the various scenarios. Each of them performs 100_000 simple queries similar to the one used in this section:

```
Not preparing the query, using `Direct_Cursor`:
   4.05s

Not preparing the query, using `Forward_Cursor`, and only
retrieving the first row:
   3.69s

Preparing the query on the client (`On_Server => False`),
with a `Direct_Cursor`. This saves the whole `GNATCOLL.SQL`
manipulations and allocations:
   2.50s

Preparing the query on the server, using `Direct_Cursor`:
   0.55s

Caching the query locally (`Use_Cache => True`):
   0.13s
```

1.10 Getting results

Once you have executed a *SELECT* query, you generally need to examine the rows that were returned by the database server. This is done in a loop, as in:

```
while Has_Row (R) loop
    Put_Line ("Max priority=" & Integer_Value (R, 0)'Img
              & " for category=" & Value (R, 1));
    Next (R);
end loop;
```

You can only read one row at a time, and as soon as you have moved to the next row, there is no way to access a previously fetched row. This is the greatest common denominator between the various database systems. In particular, it proves efficient, since only one row needs to be kept in memory at any point in time.

For each row, we then call one of the *Value* or **Value* functions which return the value in a specific row and a specific column.

We mentioned earlier there was no way to go back to a row you fetched previously except by executing the query again. This is in fact only true if you use a *Forward_Cursor* to fetch the results.

But GNATColl provides another notion, a *Direct_Cursor*. In this case, it fetches all the rows in memory when the query executes (thus it needs to allocate more memory to save every thing, which can be costly if the query is big). This behavior is supported natively by *PostgreSQL*, but doesn't exist with *sqlite*, so GNATColl will simulate it as efficiently as possible. But it will almost always be faster to use a *Forward_Cursor*.

In exchange for this extra memory overhead, you can now traverse the list of results in both directions, as well as access a specific row directly. It is also possible to know the number of rows that matched (something hard to do with a *Forward_Cursor* since you would need to traverse the list once to count, and then execute the query again if you need the rows themselves).

Direct_Cursor, produced from prepared statements, could be indexed by the specified field value and routine Find could set the cursor position to the row with specified field value.:

```
--  Prepared statement should be declared on package level.

Stmt : Prepared_Statement :=
  Prepare ("select Id, Name, Address from Contact order by Name"
           Use_Cache => True, Index_By => Field_Index'First);

procedure Show_Contact (Id : Integer) is
   CI : Direct_Cursor;
begin
   CI.Fetch (DB, Stmt);
   CI.Find (Id); -- Find record by Id

   if CI.Has_Row then
      Put_Line ("Name " & CI.Value (1) & " Address " & CI.Value (2));
   else
      Put_Line ("Contact id not found.");
   end if;
end Show_Contact;
```

In general, the low-level DBMS C API use totally different approaches for the two types of cursors (when they even provide them). By contrast, GNATColl makes it very easy to change from one to the other just by changing the type of a the result variable. So you would in general start with a *Forward_Cursor*, and if you discover you in fact need more advanced behavior you can pay the extra memory cost and use a *Direct_Cursor*.

For both types of cursors, GNATColl automatically manages memory (both on the client and on the DBMS), thus providing major simplification of the code compared to using the low-level APIs.

1.11 Creating your own SQL types

GNATColl comes with a number of predefined types that you can use in your queries. `gnatcoll_db2ada` will generate a file using any of these predefined types, based on what is defined in your actual database.

But sometimes, it is convenient to define your own SQL types to better represent the logic of your application. For instance, you might want to define a type that would be for a *Character* field, rather than use the general *SQL_Field_Text*, just so that you can write statements like:

```
declare
   C : Character := 'A';
   Q : SQL_Query;
begin
   Q := SQL_Select (.., Where => Table.Field = C);
end
```

This is fortunately easily achieved by instantiating one generic package, as such:

```
with GNATCOLL.SQL_Impl; use GNATCOLL.SQL_Impl;

function To_SQL (C : Character) return String is
begin
   return "'" & C & "'";
end To_SQL;
```

(continues on next page)

(continued from previous page)

```
package Character_Fields is new Field_Types (Character, To_SQL);
type SQL_Field_Character is new Character_Fields.Field
   with null record;
```

This automatically makes available both the field type (which you can use in your database description, as `gnatcoll_db2ada` would do, but also all comparison operators like <, >, =, and so on, both to compare with another character field, or with *Character* Ada variable. Likewise, this makes available the assignment operator = so that you can create *INSERT* statements in the database.

Finally, the package *Character_Fields* contain other generic packages which you can instantiate to bind SQL operators and functions that are either predefined in SQL and have no equivalent in GNATColl yet, or that are functions that you have created yourself on your DBMS server.

See the specs of *GNATCOLL.SQL_Impl* for more details. This package is only really useful when writing your own types, since otherwise you just have to use *GNATCOLL.SQL* to write the actual queries.

See also *GNATCOLL.SQL_Fields* for an example on how to have a full integration with other parts of *GNATCOLL.SQL*.

1.12 Query logs

The *GNATCOLL.Traces* package provides facilities to add logging. The database interface uses this module to log the queries that are sent to the server.

If you activate traces in your application, the user can then activate one of the following trace handles to get more information on the exchange that exists between the database and the application. As we saw before, the output of these traces can be sent to the standard output, a file, the system logs,...

The following handles are provided:

- SQL.ERROR This stream is activated by default. Any error returned by the database (connection issues, failed transactions,...) will be logged on this stream
- SQL This stream logs all queries that are not SELECT queries, i.e. mostly all queries that actually modify the database
- SQL.SELECT This stream logs all select queries. It is separated from SQL because very often you will be mostly interested in the queries that impact the database, and logging all selects can generate a lot of output.

In our library example, we would add the following code to see all SQL statements executed on the server:

```
with GNATCOLL.Traces;  use GNATCOLL.Traces;
procedure Main is
begin
   GNATCOLL.Traces.Parse_Config_File (".gnatdebug");
   ... --  code as before
   GNATCOLL.Traces.Finalize;  --  reclaim memory
```

and then create a .gnatdebug in the directory from which we launch our executable. This file would contain a single line containing "+" to activate all log streams, or the following to activate only the subset of fields related to SQL:

```
SQL=yes
SQL.SELECT=yes
SQL.LITE=yes
```

1.13 Writing your own cursors

The cursor interface we just saw is low-level, in that you get access to each of the fields one by one. Often, when you design your own application, it is better to abstract the database interface layer as much as possible. As a result, it is often better to create record or other Ada types to represent the contents of a row.

Fortunately, this can be done very easily based on the API provided by *GNATCOLL.SQL*. Note that *GNATCOLL.SQL.ORM* provides a similar approach based on automatically generated code, so might be even better. But it is still useful to understand the basics of providing your own objects.

Here is a code example that shows how this can be done:

```
type Customer is record
   Id   : Integer;
   First, Last : Unbounded_String;
end record;

type My_Cursor is new Forward_Cursor with null record;
function Element (Self : My_Cursor) return My_Row;
function Do_Query (DB, ...) return My_Cursor;
```

The idea is that you create a function that does the query for you (based on some parameters that are not shown here), and then returns a cursor over the resulting set of rows. For each row, you can use the *Element* function to get an Ada record for easier manipulation.

Let's first see how these types would be used in practice:

```
declare
  C : My_Cursor := Do_Query (DB, ...);
begin
  while Has_Row (C) loop
     Put_Line ("Id = " & Element (C).Id);
     Next (C);
  end loop;
end;
```

So the loop itself is the same as before, except we no longer access each of the individual fields directly. This means that if the query changes to return more fields (or the same fields in a different order for instance), the code in your application does not need to change.

The specific implementation of the subprograms could be similar to the following subprograms (we do not detail the writing of the SQL query itself, which of course is specific to your application):

```
function Do_Query return My_Cursor is
   Q : constant SQL_Query := ....;
   R : My_Cursor;
begin
   R.Fetch (DB, Q);
   return R;
end Do_Query;

function Element (Self : My_Cursor) return My_Row is
begin
  return Customer'
     (Id    => Integer_Value (Self, 0),
      First => To_Unbounded_String (Value (Self, 1)),
```

(continues on next page)

(continued from previous page)

```
      Last  => To_Unbounded_String (Value (Self, 2)));
end Element;
```

There is one more complex case though. It might happen that an element needs access to several rows to fill the Ada record. For instance, if we are writing a CRM application and query the contacts and the companies they work for, it is possible that a contact works for several companies. The result of the SQL query would then look like this:

```
contact_id | company_id
    1      |    100
    1      |    101
    2      |    100
```

The sample code shown above will not work in this case, since Element is not allowed to modify the cursor. In such a case, we need to take a slightly different approach:

```
type My_Cursor is new Forward_Cursor with null record;
function Do_Query return My_Cursor; --  as before
procedure Element_And_Next
   (Self : in out My_Cursor; Value : out My_Row);
```

where *Element_And_Next* will fill Value and call Next as many times as needed. On exit, the cursor is left on the next row to be processed. The usage then becomes:

```
while Has_Row (R) loop
   Element_And_Next (R, Value);
end loop;
```

To prevent the user from using Next incorrectly, you should probably override *Next* with a procedure that does nothing (or raises a Program_Error maybe). Make sure that in *Element_And_Next* you are calling the inherited function, not the one you have overridden, though.

There is still one more catch. The user might depend on the two subprograms *Rows_Count* and *Processed_Rows* to find out how many rows there were in the query. In practice, he will likely be interested in the number of distinct contacts in the tables (2 in our example) rather than the number of rows in the result (3 in the example). You thus need to also override those two subprograms to return correct values.

1.14 The Object-Relational Mapping layer (ORM)

GNATColl provides a high-level interface to manipulate persistent objects stored in a database, using a common paradigm called an object-relational mapping. Such mappings exist for most programming languages. In the design of GNATColl, we were especially inspired by the python interface in *django* and *sqlalchemy*, although the last two rely on dynamic run time introspection and GNATColl relies on code generation instead.

This API is still compatible with *GNATCOLL.SQL*. In fact, we'll show below cases where the two are mixed. It can also be mixed with *GNATCOLL.SQL.Exec*, although this might be more risky. Communication with the DBMS is mostly transparent in the ORM, and it uses various caches to optimize things and make sure that if you modify an element the next query(ies) will also return it. If you use *GNATCOLL.SQL.Exec* directly you are bypassing this cache so you risk getting inconsistent results in some cases.

In ORM, a table is not manipulated directly. Instead, you manipulate objects that are read or written to a table. When we defined our database schema (*Database schema*), we gave two names on the first line of a table definition. There was the name of the table in the database, and the name of the object that each row represent. So for our library example we have defined *Customer*, *Book* and *Dvd* objects. These objects are declared in a package generated automatically by *gnatcoll_db2ada*.

There is first one minor change we need to do to our library example. The ORM currently does not handle properly cases where an abstract class has foreign keys to other tables. So we remove the *borrowed_by* field from the *Media* table, and change the *books* table to be:

```
| TABLE (media) | books                          | book |     | The books in the library |
| pages         | INTEGER                        |      | 100 |                          |
| borrowed_by   | FK customers(borrowed_books) | NULL |     | Who borrowed the media   |
```

Let's thus start by generating this code. We can replace the command we ran earlier (with the *-api* switch) with one that will also generate the ORM API:

```
gnatcoll_db2ada -dbmode dbschema.txt -api Database -orm ORM
```

The ORM provides a pool of database connections through the package *GNATCOLL.SQL.Sessions*. A session therefore acts as a wrapper around a connection, and provides a lot more advanced features that will be described later. The first thing to do in the code is to configure the session pool. The *Setup* procedure takes a lot of parameters to make sessions highly configurable. Some of these parameters will be described and used in this documentation, others are for special usage and are only documented in `gnatcoll-sql-sessions.ads`. Here will we use only specify the mandatory parameters and leave the default value for the other parameters:

```
GNATCOLL.SQL.Sessions.Setup
   (Descr  => GNATCOLL.SQL.Sqlite.Setup ("library.db"),
    Max_Sessions => 2);
```

The first parameter is the same *Database_Description* we saw earlier (*Connecting to the database*), but it will be freed automatically by the sessions package, so you should not free it yourself.

Once configure, we can now request a session. Through a session, we can perform queries on the database, make objects persistent, write the changes back to the database,.... We configured the session pool to have at most 2 sessions. The first time we call *Get_New_Session*, a new session will be created in the pool and marked as busy. While you have a reference to it in your code (generally as a local variable), the session belongs to this part of the code. When the session is no longer in scope, it is automatically released to the pool to be reused for the next call to *Get_New_Session*. If you call *Get_New_Session* a second time while some part of your code holds a session (for instance in a different task), a new session will be created. But if you do that a third time while the other two are busy, the call to *Get_New_Session* is blocking until one of the two sessions is released to the pool.

This technique ensures optimal use of the resources: we avoid creating a new session every time (with the performance cost of connecting to the database), but also avoid creating an unlimited number of sessions which could saturate the server. Since the sessions are created lazily the first time they are needed, you can also configure the package with a large number of sessions with a limited cost.

Let's then take a new session in our code:

```
Session : constant Session_Type := Get_New_Session;
```

and let's immediately write our first simple query. A customer comes at the library, handles his card and we see his id (1). We need to look up in the database to find out who he is. Fortunately, there is no SQL to write for this:

```
C : ORM.Detached_Customer'Class := Get_Customer (Session, Id => 1);
```

The call to *Get_Customer* performs a SQL query transparently, using prepared statements for maximum efficiency. This results in a *Customer* object.

ORM is the package that was generated automatically by *gnatcoll_db2ada*. For each table in the database, it generates a number of types:

- *Customer*

This type represents a row of the *Customers* table. It comes with a number of primitive operations, in particular one for each of the fields in the table. Such an object is returned by a cursor, similarly to what was described in the previous section (*Writing your own cursors*). This object is no longer valid as soon as the cursor moves to the next row (in the currently implementation, the object will describe the next row, but it is best not to rely on this). As a benefit, this object is light weight and does not make a copy of the value of the fields, only reference the memory that is already allocated for the cursor.

This object redefines the equality operator ("=") to compare the primary key fields to get expected results.

- *Detached_Customer*

  A detached object is very similar to the *Customer* object, but it will remain valid even if the cursor moves or is destroyed. In fact, the object has made a copy of the value for all of its fields. This object is heavier than a *Customer*, but sometimes easier to manager. If you want to store an object in a data structure, you must always store a detached object.

  A detached object also embeds a cache for its foreign keys. In the context of our demo for instance, a *Book* object was borrowed by a customer. When returning from a query, the book knows the id of that customer. But if call *B.Borrowed_By* this returns a *Detached_Customer* object which is cached (the first time, a query is made to the DBMS to find the customer given his id, but the second time this value is already cached).

  One cache create a *Detached_Customer* from a *Customer* by calling the *Detach* primitive operation.

- *Customer_List*

  This type extends a *Forward_Cursor* (*Getting results*). In addition to the usual *Has_Row* and *Next* operations, it also provides an *Element* operation that returns a *Customer* for easy manipulation of the results.

- *Direct_Customer_List*

  This type extends a *Direct_Cursor*. It also adds a *Element* operation that returns a *Customer* element.

- *Customers_Managers*

  This type is the base type to perform queries on the DBMS. A manager provides a number of primitive operations which end up creating a SQL query operation in the background, without making that explicit.

  Let's first write a query that returns all books in the database:

```
declare
   M : Books_Managers := All_Books;
   BL : Book_List := M.Get (Session);
   B : Book;
begin
   while BL.Has_Row loop
      B := BL.Element;
      Put_Line ("Book: " & B.Title);
      Put_Line ("   Borrowed by: " & B.Borrowed_By.Last);
      BL.Next;
   end loop;
end;
```

  The manager *M* corresponds to a query that returns all the books in the database. The second line then executes the query on the database, and returns a list of books. We then traverse the list. Note how we access the book's title by calling a function, rather than by the index of a field as we did with *GNATCOLL.SQL.Exec* with Value(B, 0). The code is much less fragile this way.

  The line that calls *Borrowed_By* will execute an additional SQL query for each book. This might be inefficient if there is a large number of books. We will show later how this can be optimized.

The manager however has a lot more primitive operations that can be used to alter the result. Each of these primitive operations returns a modified copy of the manager, so that you can easily chain calls to those primitive operations. Those operations are all declared in the package *GNATCOLL.SQL.ORM.Impl* if you want to look at the documentation. Here are those operations:

- *Get* and *Get_Direct*

  As seen in the example above, these are the two functions that execute the query on the database, and returns a list of objects (respectively a *Customer_List* and a *Direct_Customer_List*).

- *Distinct*

  Returns a copy of the manager that does not return twice a row with the same data (in SQL, this is the "DISTINCT" operator)

- *Limit* (Count : Natural; From : Natural := 0)

  Returns a copy of the manager that returns a subset of the results, for instance the first *Count* ones.

- *Order_By* (By : SQL_Field_List)

  Returns a copy of the manager that sorts the results according to a criteria. The criteria is a list of field as was defined in *GNATCOLL.SQL*. We can for instance returns the list of books sorted by title, and only the first 5 books, by replacing *M* with the following:

```
M : Books_Managers := All_Books.Limit (5).Order_By (Books.Title);
```

- *Filter*

  Returns a subset of the result matching a criteria. There are currently two versions of Filter: one is specialized for the table, and has one parameter for each field in the table. We can for instance return all the books by Alexandre Dumas by using:

```
M : Books_Managers := All_Books.Filter (Author => "Dumas");
```

  This version only provides the equality operator for the fields of the table itself. If for instance we wanted all books with less than 50 pages, we would use the second version of filter. This version takes a *GNATCOLL.SQL.SQL_Criteria* similar to what was explained in previous sections, and we would write:

```
M : Books_Managers := All_Books.Filter (Condition => Books.Pages < 50);
```

  More complex conditions are possible, involving other tables. Currently, the ORM does not have a very user-friendly interface for those, but you can always do this by falling back partially to SQL. For instance, if we want to retrieve all the books borrowed by user "Smith", we need to involve the *Customers* table, and thus make a join with the *Books* table. In the future, we intend to make this join automatic, but for now you will need to write:

```
M : Books_Managers := All_Books.Filter
  (Books.FK (Customers)
   and Customers.Last = "Smith");

-- SQL query: SELECT books.pages, books.borrowed_by, books.id,
--    books.title, books.author, books.published
--    FROM books, customers
--    WHERE books.borrowed_by=customers.id AND customers.last='Smith'
```

  This is still simpler code than we were writing with *GNATCOLL.SQL* because we do not have to specify the fields or tables, and the results are objects rather than fields with specific indexes.

– *Select_Related* (Depth : Integer; Follow_Left_Join : Boolean)

This function returns a new manager that will retrieve all related objects. In the example we gave above, we mentioned that every time *B.Borrowed_By* was called, this resulted in a call to the DBMS. We can optimize this by making sure the manager will retrieve that information. As a result, there will be a single query rather than lots. Be careful however, since the query will return more data, so it might sometimes be more efficient to perform multiple smaller queries.

Depth indicates on how many levels the objects should be retrieved. For instance, assume we change the schema such that a Book references a Customer which references an Address. If we pass 1 for *Depth*, the data for the book and the customer will be retrieved. If however you then call *B.Borrowed_By.Address* this will result in a query. So if you pass 2 for *Depth* the data for book, customers and addresses will be retrieved.

The second parameter related to efficiency. When a foreign key was mentioned as *NOT NULL* in the schema, we know it is always pointing to an existing object in another table. *Select_Related* will always retrieve such objects. If, however, the foreign key can be null, i.e. there isn't necessarily a corresponding object in the other table, the SQL query needs to use a *LEFT JOIN*, which is less efficient. By default, GNATColl will not retrieve such fields unless *Follow_Left_Join* was set to True.

In our example, a book is not necessarily borrowed by a customer, so we need to follow the left joins:

```
M : Books_Managers := All_Books.Filter
  (Books.FK (Customers)
   and Customers.Last = "Smith")
  .Select_Related (1, Follow_Left_Join => True);

-- SQL query:  SELECT books.pages, books.borrowed_by, books.id,
--    books.title, books.author, books.published,
--    customers.id, customers.first, customers.last
--    FROM (books LEFT JOIN customers ON books.borrowed_by=customers.id)
--    WHERE books.borrowed_by=customers.id AND customers.last='Smith'
```

1.14.1 reverse relationships

In fact, the query we wrote above could be written differently. Remember we have already queries the *Customer* object for id 1 through a call to *Get_Customer*. Since our schema specified a *reverse_name* for the foreign key *borrowed_by* in the table *books*, we can in fact simply use:

```
BL := C.Borrowed_Books.Get (Session);

-- SQL: SELECT books.pages, books.borrowed_by, books.id, books.title,
--    books.author, books.published FROM books
--    WHERE books.borrowed_by=1
```

Borrowed_Books is a function that was generated because there was a *reverse_name*. It returns a *Books_Managers*, so we could in fact further filter the list of borrowed books with the same primitive operations we just saw. As you can see, the resulting SQL is optimal.

Let's optimize further the initial query. We have hard-coded the customer name, but in fact we could be using the same subprograms we were using for prepared statements (*Prepared queries*), and even prepare the query on the server for maximum efficiency. Since our application is likely to use this query a lot, let's create a global variable:

```
M : constant Books_Managers := All_Books.Filter
  (Books.FK (Customers)
   and Customers.Id = Integer_Param (1))
```

(continues on next page)

(continued from previous page)

```
  .Select_Related (1, Follow_Left_Join => True);

MP : constant ORM_Prepared_Statement :=
  M.Prepare (On_Server => True);

... later in the code

Smith_Id : constant Natural := 1;
BL : Book_List := MP.Get (Session, Params => (1 => Smith_Id));
```

The last call to *Get* is very efficient, with timing improvements similar to the ones we discussed on the session about prepared statements (*Prepared queries*).

1.15 Modifying objects in the ORM

The ORM is much more than writing queries. Once the objects are persistent, they can also be simplify modified, and they will be saved in the database transparently.

Let's start with a simple example. In the previous section, we retrieve an object *C* representing a customer. Let's change his name, and make sure the change is in the database:

```
C := Get_Customer (Session, 1);
C.Set_Last ("Smith");
C.Set_First ("Andrew");
Session.Commit;
```

A reasonable way to modify the database. However, this opens a can of complex issues that need to be dealt with.

When we called *Set_Last*, this modify the objects in memory. At this point, printing the value of *C.Last* would indeed print the new value as expected. The object was also marked as modified. But no change was made in the database.

Such a change in the database might in fact be rejected, depending on whether there are constraints on the field. For instance, say there existed a constraint that *Last* must be the same *First* (bear with me, this is just an example). If we call *Set_Last*, the constraint is not satisfied until we also call *Set_First*. But if the former resulted in an immediate change in the database, it would be rejected and we would not even get a change to call *Set_First*.

Instead, the session keeps a pointer to all the objects that have been modified. When it is committed, it traverses this list of objects, and commits their changes into the database. In the example we gave above, the call to *Commit* will thus commit the changes to *C* in the database. For efficiency, it uses a single SQL statement for that, which also ensures the constraint remains valid:

```
UPDATE customers SET first='Andrew', last='Smith' WHERE customers.id=1;
```

We can create a new customer by using similar code:

```
C := New_Customer;
C.Set_First ("John");
C.Set_Last ("Lee");
Session.Persist (C);

Session.Commit;
```

New_Customer allocates a new object in memory. However, this object is not persistent. You can call all the *Set_** subprograms, but the object will not be saved in the database until you add it explicitly to a session with a call to *Persist*, and then *Commit* the session as usual.

Another issue can occur when objects can be modified in memory. Imagine we retrieve a customer, modify it in memory but do not commit to the database yet because there are other changes we want to do in the same SQL transaction. We then retrieve the list of all customers. Of course, the customer we just modified is part of this list, but the DBMS does not know about the change which currently only exists in memory.

Thankfully, GNATColl takes care of this issue automatically: as we mentioned before, all modified objects are stored in the session. When traversing the list of results, the cursors will check whether the session already contains an element with the same id that it sees in the result, and if yes will return the existing (i.e. modified) element. For instance:

```
C := Get_Customer (Session, Id => 1);
C.Set_Last ("Lee");

CL : Customer_List := All_Customers.Get (Session);
while CL.Has_Row loop
   Put_Line (CL.Element.Last);
   CL.Next;
end loop;
```

The above example uses *CL.Element*, which is a light-weight *Customer* object. Such objects will only see the in-memory changes if you have set *Flush_Before_Query* to true when you configured the sessions in the call to *GNATCOLL.SQL.Sessions.Setup*. Otherwise, it will always return what's really in the database.

If the example was using *Detached_Customer* object (by calling *CL.Element.Detach* for instance) then GNATColl looks up in its internal cache and returns the cached element when possible. This is a subtlety, but this is because an *Customer* only exists as long as its cursor, and therefore cannot be cached in the session. In practice, the *Flush_Before_Query* should almost always be true and there will be not surprising results.

1.16 Object factories in ORM

Often, a database table is used to contain objects that are semantically of a different kind. In this section, we will take a slightly different example from the library. We no longer store the books and the dvds in separate tables. Instead, we have one single *media* table which contains the title and the author, as well as a new field *kind* which is either 0 for a book or 1 for a dvd.

Let's now look at all the media borrowed by a customer:

```
C : constant Customer'Class := Get_Customer (Session, Id => 1);
ML : Media_List := C.Borrowed_Media.Get (Session);

while ML.Has_Row loop
   case ML.Element.Kind is
      when 0 =>
         Put_Line ("A book " & ML.Element.Title);
      when 1 =>
         Put_Line ("A dvd " & ML.Element.Title);
   end case;
   ML.Next;
end loop;
```

This code works, but requires a case statement. Now, let's imagine the check out procedure is different for a book and a DVD (for the latter we need to check that the disk is indeed in the box). We would have two subprograms *Checkout_Book* and *Checkout_DVD* and call them from the case. This isn't object-oriented programming.

Instead, we will declare two new types:

```
type My_Media is abstract new ORM.Detached_Media with private;
procedure Checkout (Self : My_Media) is abstract;

type Detached_Book is new My_Media with private;
overriding Checkout (Self : Detached_Book);

type Detached_DVD is new My_Media with private;
overriding Checkout (Self : Detached_DVD);
```

We could manually declare a new Media_List and override *Element* so that it returns either of the two types instead of a *Media*. But then we would also need to override *Get* so that it returns our new list. This is tedious.

We will instead use an element factory in the session. This is a function that gets a row of a table (in the form of a *Customer*), and returns the appropriate type to use when the element is detached (by default, the detached type corresponding to a *Customer* is a *Detached_Customer*, and that's what we want to change).

So let's create such a factory:

```
function Media_Factory
   (From    : Base_Element'Class;
    Default : Detached_Element'Class) return Detached_Element'Class
is
begin
   if From in Media'Class then
      case Media (From).Kind is
         when 0 =>
            return R : Detached_Book do null; end return;
         when 1 =>
            return R : Detached_DVD do null; end return;
         when others =>
            return Default;
      end case;
   end if;
   return Default;
end Media_Factory;

Session.Set_Factory (Media_Factory'Access);
```

This function is a bit tricky. It is associated with a given session (although we can also register a default factory that will be associated with all sessions by default). For all queries done through this session (and for all tables) it will be called. So we must first check whether we are dealing with a row from the *Media* table. If not, we simply return the suggested *Default* value (which has the right *Detached_** kind corresponding to the type of *From*).

If we have a row from the *Media* table, we then retrieve its kind (through the usual automatically generated function) to return an instance of *Detached_Book* or *Detached_DVD*. We use the Ada05 notation for extended return statements, but we could also use a declare block with a local variable and return that variable. The returned value does not need to be further initialized (the session will take care of the rest of the initialization).

We can now write our code as such:

```
C : constant Customer'Class := Get_Customer (Session, Id => 1);
ML : Media_List := C.Borrowed_Media.Get (Session);

while ML.Has_Row loop
   Checkout (ML.Element.Detach);   --  Dispatching
```

(continues on next page)

(continued from previous page)

```
   ML.Next;
end loop;
```

The loop is cleaner. Of course, we still have the case statement, but it now only exists in the factory, no matter how many loops we have or how many primitive operations of the media we want to define.

CHAPTER

TWO

XREF: CROSS-REFERENCING SOURCE CODE

When manipulating source code, programmers needs to know where the various symbols are defined, where they are used, and so on. This is generally available directly from their IDE and editors. But computing this information in the first place is tricky, especially for languages that support overloading of subprograms.

Some compilers like GNAT and gcc can generate this information for Ada code bases. For instance, GNAT will generate `.ali` files, which contain the navigation information, when compiling Ada or SPARK code.

`GNATCOLL.Xref` can then be used to parse and aggregate all those files into a single sqlite database, which can be conveniently used to answer queries such as "give me the declaration for this entity", "list all places where this entity is used", "show all subprograms that could be called in practice at this dispatching call", "what files does this file depend on", "show me the call graph for this application",...

To use this package, some initialization needs to be performed first:

```
with GNATCOLL.Xref;     use GNATCOLL.Xref;
with GNATCOLL.SQL.Sqlite;
with GNATCOLL.Projects; use GNATCOLL.Projects;   --  1
with GNATCOLL.VFS;      use GNATCOLL.VFS;
with GNAT.Strings;

procedure Support is
   DB : Xref_Database;
   Tree : Project_Tree_Access := new Project_Tree;
   Error : GNAT.Strings.String_Access;

begin
   Tree.Load (Create ("prj.gpr"));  --  2
   Setup_DB
      (DB, Tree,
       GNATCOLL.SQL.Sqlite.Setup (Database => "testdb.db"),
       Error);  -- 3
   Free (Error);
   Parse_All_LI_Files (DB, Tree.Root_Project);   --  4
end Support;
```

GNATCOLL needs to be able to find the `*li` files. For this, it depends on project files (as supported by `GNATCOLL.Projects`). So the first thing to do is to parse the project (step 2).

We then need to tell GNATCOLL where the cross-reference information need to be aggregated. In this example, it will be stored in a sqlite database on the disk. By using a name ":memory:" instead, we would create a temporary in-memory database. This is in general faster, but uses more memory and needs to be recreated every time the program is restarted. We could also decide to store the information in any other database supported by `GNATCOLL.SQL.Exec`, for instance PostgreSQL.

Finally, in step 4 we let GNATCOLL parse all the `*li` files that are relevant for this project. This operation can take a while, depending on the size of the project. However, if the database already exists on the disk, it will simply be updated by parsing the files that are not already up-to-date. When all files are up-to-date, this operation is almost immediate.

At this point, we now have a database that we can start querying. Here are a few examples, but see the documentation `gnatcoll-xref.ads` for more types of queries. All these queries have a similar API: they return a **cursor** which iterates over the result returned by a SQL query. There are various kinds of cursors, depending on whether they return files, entities, or references to entities. But they all support the *Has_Element*, *Element* and *Next* operations, so all loops will look similar:

```
pragma Ada_05;   --  use object-dotted-notation
with GNATCOLL.VFS;   use GNATCOLL.VFS;

declare
    Entity : Entity_Information;
    Ref    : Entity_Reference;
    File   : Virtual_File;
    Refs   : References_Cursor;
begin
    File := Tree.Create ("source.ads");   --  5
    Ref := DB.Get_Entity ("Method", File, Line => 2);   --  6
    Entity := Ref.Entity;

    DB.References (Entity, Refs);   --  7
    while Refs.Has_Element loop
       Ref := Refs.Element;
       Put_Line (" at " & Ref.File.Display_Full_Name & ':'
          & Ref.Line'Img & ':' & Ref.Column'Img);
       Refs.Next;
    end loop;
end;
```

This example will print all the references to the entity that is referenced in file source.ads at line 2 (the column is unspecified).

Step 5 gets a handle on the source file. Here, we depend on the project to find the precise directory in which the source file is located. We can of course use an absolute file name instead.

Step 6 gets handle on the entity referenced on line 2 in this file. Such an entity is the starting point for most queries defined in *GNATCOLL.Xref*.

Finally, on step 7 and the loop below we iterate over all references, and print their location on the standard output.

Let's do a much more complex query: we want to see all references to that entity, but also places where the entity might be called through a *renames* statement, or called through a dispatching call via an overriding method defined on a child tagged type (assuming this is a primitive operation of a tagged type in the first place). We also want to see all locations where a method that overrides "Method" is called:

```
declare
   Refs : Recursive_References_Cursor;
begin
   DB.Recursive (Entity, GNATCOLL.Xref.References'Access,
                 From_Overriding => True, From_Overridden => True,
                 From_Renames => True);
   while Refs.Has_Element loop
        ... same as before
```

(continues on next page)

(continued from previous page)

```
         Refs.Next;
   end loop;
end;
```

As shown above, the programing pattern is always the same.

GNATCOLL.Xref provides many more subprogram to get information like the list of fields for a record type (or a C structure), the list of primitive operations or methods for a tagged object or a class, the call graph for a subprogram,...

It is also able to extract documentation for an entity from the source code, by looking at the lines of code just before or just after the declaration or the body of the entity.

CHAPTER

THREE

XREF: GNATINSPECT

As discussed in the previous section, GNATCOLL provides an Ada API to perform cross-references queries.

There exist a few alternatives when you want to reuse that cross-reference information from other tools, or command line scripts.

You can of course access the sqlite database directly. Most programming languages have an interface to sqlite. For instance python does.

But GNATCOLL provides a command line tool dedicated to that purpose, named **gnatinspect**.

When it is first started on a project, this tool will refresh the xref database by parsing all the ALI files from the project. This might take a while (up to several minutes) the first time, unless of course the xref were already up-to-date because you had loaded the project in GPS first, or already run gnatinspect.

gnatinspect then displays an interactive prompt that lets you perform various queries on the database. The full list of queries is available by typing "help" at the prompt, but this documentation will demonstrate some of them.

Let's first look at a number of command line switches that might be useful:

- *–db=ARG*: this switch can be used to specify the name of the database.

  By default, gnatinspect checks in the project whether there exists an attribute IDE'Xref_Database, which should specify a file name (relative to the project's object_dir) for the database.

  If this attribute does not exist, it defaults to "gnatinspect.db" in the project's object directory.

  If there is no object directory defined in the project, the file is created in the project's directory itself. You can however specify any name, including an absolute path, or a path relative to the project's object directory.

  An alternative is to specify ':memory:', which creates the database in memory. This is of course a temporary database which will disappear when gnatinspect exits, and cannot be shared with other tools.

- *–nightlydb=ARG*: this switch can help speed up the initial startup of gnatinspect. The idea is that in a lot of cases, the software on which a team works is build nightly in a common setup. Running gnatinspect in that setup will create or update an xref database. Individual developers can then create their own copy of the database by starting from the contents of the nightly database (which is pointed to by the –nightlydb switch), and then gnatinspect will parse the ALI files in the user's setup that are different from the nightly ones.

- *–runtime*: by default, gnatinspect will only parse the ALI files from your project (and of course the ones from imported projects). It will not however parse the ALI files found in predefined directories, like for instance the GNAT runtime. This saves time in general. If you click on a call to one of the runtime subprograms in your own code, gnatinspect will be able to point you to its declaration. However, you will not have access to the body, because the link from declaration to body is found in the ALI files of the runtime.

- *–command=ARG*: gnatinspect will update the xref database as usual, then execute a command, display its result, and exit. This can be convenient when calling gnatinspect from another tool, like Emacs or vi.

- *–file=ARG*: similar to –command, but reads the commands to execute from a file. The file can contain comments (starting with '–'. See also the –lead switch.
- *–lang=LANG:SPEC:BODY:OBJ*: specifies a naming scheme for a language. The preferred approach is to use a configuration project file (such as those generated by gprconfig for instance), that would define attributes such as *Naming'Spec_Suffix*, *Naming'Body_Suffix* and *Compiler'Object_File_Suffix*. However, this switch provides an alternative whereby you can specify the same values directly on the command line. For instance, the equivalent of:

```
configuration project Autoconf is
   package Naming is
      for Spec_Suffix ("MyLang") use ".myl";
   end Naming;
   package Compiler is
      for Object_File_Suffix ("MyLang") use ".ali";
   end Compiler;
end Autoconf;
```

  is to use:

```
--lang=MyLang:.myl::.ali
```

  A third alternative is to have the same contents as the configuration project file above, directly in your own project file. This has the same effect, but needs to be duplicated in each of your project file.

  Given one of the above, and assuming your project file includes:

```
for Languages use ("Ada", "MyLang");
```

  then any file with the `.myl` extension will be correctly detected by the project manager, and any `.ali` file with the same base name will be parsed by gnatinspect to find cross-reference information. Remember that the switch *–config=autoconf.cgpr* must be passed to gnatinspect if the information is provided via a config project file.
- *–lead=ARG* should be used in coordination with –file, and specify lines to ignore from the file. All lines starting with the given prefix will be ignored.
- *–basenames*: controls the display of file names in the output. By default, gnatinspect outputs full path information.
- *–exit*: if this switch is specified, gnatinspect updates the xref database and exits immediately.
- *–project=ARG* or *-P ARG* specifies the name of the project to load. This switch is mandatory.
- *-X VAR=VALUE* is used to specify the value of scenario variables used in your project. This is similar to the homonym switch in gprbuild.
- *–symlinks* should be specified if your project uses symbolic links for files. This will ensure that the links are fully resolved as stored in the database, and thus that when a file is visible through different links, the information is appropriately coalesced in the database for that file.
- *–subdirs=ARG* is similar to the homonym switch in gprbuild
- *–tracefile=ARG* is used to point to a file compatible with GNATCOLL.Traces that controls the debug information generated by gnatinspect. By default, gnatinspect parses a file called '.gnatdebug' in the current directory.
- *–encoding=ARG* is the character encoding used for source and ALI files. By default, gnatinspect assumes they are encoded in UTF-8.

Once it has finished parsing the xref information, gnatinspect displays an interactive prompt, where a number of commands can be used to perform queries. In a lot of cases, these commands take some file information as argument (either just the file, or an entity name and the file in which it is defined).

The file names can be given as either a base name, or relative to the current directory, or even a full name. But file names are ambiguous (even when a full path is specified) when aggregate projects are used. It is valid for a given file to be part of multiple aggregate projects, and depending on the project we are considering the result of the xref queries might vary).

To remove the ambiguity, it is possible to specify the project to which the file belongs. The project is specified either as a project name (which itself could be ambiguous with aggregate projects), or as a full path.

In all commands below, whenever the parameter specifies ":file", you can use instead ":file:project" if there are ambiguities. It is also possible not to specify the file, in which case the entity will be looked for in all sources of the project.

Here is the full list of commands supported by gnatinspect:

- *decl name:file:line:column* is probably the most useful command. Given a reference to an entity, it will indicate where the entity is declared. The line and column informations are optional:

```
>>> decl Func:file.adb:12
Func:/some/path/file2.adb:20:9
```

- *body name:file:line:column* is similar to *decl*, but will return the location of the body of the entity. When the entity is an Ada private type, its body is in fact the location of the full declaration for that type.
- *refs name:file:line:column* displays all known references to the entity.
- *refs_overriding name:file:line:column* displays all known references to the entity or one of its overriding entities
- *doc name:file:line:column* will display documentation for the entity. The exact format for the entity might change in future versions of gnatinspect, but will in general include the type of entity, the location of its declaration, and any comment associated with it in the source code:

```
>>> doc Func:file.adb
procedure declared at /some/path/file2.adb:20:9

And the comments written below Func in file2.adb
```

- *fields name:file:line:column* displays the fields of an Ada record type or a C struct:

```
>>> fields Rec:file.ads:20
A:/some/path/file.ads:21
B:/some/path/file.ads:22
```

- *child_types name:file:line:column* lists all child types for this entity, for instance classes that inherit from the entity. This is the opposite of *parent_types*.
- *child_types_recursive name:file:line:column* is similar to *child_types* but will also list the child types of the children. This query can be used to find a whole tagged type hierarchy (or class hierarchy in C++).
- *parent_types name:file:lin:column* returns the parent types for the entity, for instance the classes or interfaces from which it derives. See also *child_types*.
- *methods name:file:line:column* returns the list of methods (or primitive operations) for the entity.
- *method_of name:file:line:column* returns the class or tagged type for which the entity is a method.
- *calls name:file:line:column* lists all entities called by the entity. This includes all entities defined within the scope of the entity (so for a subprogram this will be the list of local variables, but for a package this includes all subprograms and nested packages defined within that package).
- *callers name:file:line:column* lists all entities that call the entity. This information is also available from a call to 'refs', but 'callers' return the callers directly, instead of references to the original entity.

- *overrides name:file:line:column* returns the entity that is overridden by the entity (generally a method from a parent class).
- *overridden name:file:line:column* returns the list of entities that override the parameter (generally methods from children classes).
- *overridden_recursive name:file:line:column* returns the list of entities that override the parameter (generally methods from children classes). This is recursive.
- *type name:file:line:column* returns the type of the entity (variable or constant). For an enumeration literal, this returns the corresponding enumeration.
- *component name:file:line:column* returns the component type of the entity (for arrays for instance).
- *literals name:file:line:column* returns the valid literal values for an enumeration.
- *pointed name:file:line:column* returns the type pointed to by the entity.
- *qname name:file:line:column* returns the fully qualified name for the entity.
- *params name:file:line:column* returns the list of parameters for the subprogram.

A number of queries are related to the source files of the project:

- *importing filename* lists the files that import the file (via with statements in Ada or #include in C for instance)
- *imports filename* lists the files that the file imports (via with statements in Ada or #include in C for instance). See also *depends_on*.
- *depends filename* lists the files that the file depends on (recursively calling *imports*)
- *entities file* lists all entities referenced or declared in the file.

Finally, some commands are not related to entities or source files:

- *refresh* refreshes the contents of the xref database, by parsing all ALI files that have been changed.
- *shell* Execute a shell command (an alternative is to use '!' as the command).
- *scenario VARIABLE VALUE* changes the value of a scenario variable, and reparse the project.
- *time command arguments* executes the command as usual, and report the time it took to execute it.

CHAPTER

FOUR

INDICES AND TABLES

- genindex

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

INDEX

F

Flush_Before_Query, 29

P

projects
    aggregate projects, 38