APQ Ada95 Database Binding to PostgreSQL/MySQL

September 7, 2003

Contents

Chapter 1

Introduction

1.1 APQ Version 2.1

This manual documents APQ Version 2.1, which is released under a dual ACL and GPL (GNU Public License) arrangement. The dual license arrangement is designed to give both the distributor and user the necessary freedoms to enjoy the fair use and distribution of the sources contained in this project. See file COPYING for more details.

1.2 Supported Databases

The APQ binding was initially created to satisfy the simple need to allow Ada programs to use a PostgreSQL database. However, as Open Sourced database technologies continue to advance, the need to allow other databases to be used, becomes greater. Rather than write a unique Ada binding for each one, it was conceptualized that a common API could emerge within the APQ framework. To this end, the APQ binding has been reworked rather extensively for version 2.x, to permit increasing levels of general support of other database technologies, including MySQL.

The database technologies supported in this version of the APQ binding are:

Database	Version	SQL	Blob
PostgreSQL	1.x	Yes	Yes
MySQL	2.x	Yes	No

The above table needs some explanation:

Version is the version of APQ where the database was first supported.

SQL indicates whether the common SQL functions are supported.

Blob indicates whether blob support is present.

As the reader can observe in the table above, the support for MySQL is incomplete in APQ 2.1. The blob support is lacking in APQ for MySQL, because MySQL's blob interface is not as complete as provided by PostgreSQL. Where PostgreSQL provides the facility for virtually limitless sized blobs, a MySQL blob must fit within a "column", very much like a text field. For this reason, the facility to perform stream oriented I/O is lacking on a blob in APQ for MySQL.

1.2.1 The Future of Blob Support for MySQL

Much investigation and research is required to adequately resolve the blob issue in APQ. Rather than hold back the binding from general use, where blob functionality may have limited use anyway, it was decided to release APQ 2.0 with the common API for the two databases, and leaving the resolution of the blob API for a future release.

If you are a developer, who hopes to write portable database code, then please be aware that the PostgreSQL blob API is subject to future revision. Potentially, this could be fairly extensively revised, but every attempt will be made to leave a migration path open to the developer.

1.3 Generic Database Support

One of the main goals of the APQ version 2.0 release, was to develop a common API, that does not discriminate based upon the database technology being selected. The ideal was to allow a developer to write a procedure that would accept a classwide database objects, and perform database operations without needing to be concerned whether the database being used was PostgreSQL or MySQL. To a large extent, the author believes that this goal has been achieved.

1.3.1 Generic Limitations

It must be admited however, there are some areas where the database technologies were very different. Consequently, some exceptions and work-arounds will be required by the programmer. An example of this is that MySQL requires that all rows be fetched from a SELECT query. A failure to do this, corrupts the communication between the server and the client. Consequently, APQ works around this by defaulting to use the C library call mysql_store_result() instead of the alternative, which is mysql_use_result(). However, if the result set is large, then receiving all of the rows into the clients memory is not a suitable choice. Consequently, APQ does provide some MySQL specific ways to manage this setting.

The MySQL database software also provides the special "LIMIT row_count" extension, if the client program is only interested in the first n rows of the result. If for example, you have a price file containing stock price history, you may want to query the most recent price for it. The simplest way to do this would be to perform a SELECT on the table with a descending price date sort sequence (or index). But if you only want the first (most recent) row returned, you do not want to retrieve the entire price history into the memory of your client! This is what mysql_store_result() implies

(APQ default). So the application programmer will need to plan for this, when MySQL is used. He will need to do one of the following:

- Cause mysql_use_result() to be used instead (change the APQ default), and then fetch all of the rows, one by one.
- Use the MySQL "LIMIT 1" SQL extension to limit the results to 1 row.

The problem of course, is that this type of handling must only be done for MySQL databases. Consequently, APQ also provides an API so that the application may query which database is being used.

1.3.2 Package Structure Reorganization

When only one database product was supported, the package hierarchy was simple. To support multiple databases however, it was necessary to reorganize the package hierarchy. Additionally, it was recognized that even though the product was dubbed APQ, the top level package name was PostgreSQL. This was simply poor planning. This also lead to a possible conflict if a customer site already has a package of that name. For these reasons, the following changes were made:

- The top level package is now APQ. This matches the product name, and eliminates any potential clash with a PostreSQL Ada binding that the PostgreSQL people may someday release.
- The PostgreSQL support has been moved to APQ.PostgreSQL.
- Client support has moved to APQ.PostgreSQL.Client.
- The MySQL support has been added to APQ.MySQL and APQ.MySQL.Client.

With the new organization, other database products like Oracle, SyBase and DB2 are possible at some future release.

1.3.3 Type Name Reorganization

In addition to package names, it was quickly realized that a PostgreSQL specific type name PG_Boolean didn't seem appropriate in a MySQL context. Consequently, the naming conventions for data types have migrated from a PG_ prefix, to a more generic APQ_ prefix instead. The package APQ.PostgreSQL will maintain subtype equivalence definitions for type names, to ease migration of existing PostgreSQL programs to the newer versions of APQ. However, the programmer is strongly advised to revise existing programs where possible.

The only completely renamed data type was the renaming of PG_Oid to Row_ID_Type. Again, the subtype equivalence is available in APQ.PostgreSQL, to ease the migration to the new APQ versions.

1.4 The APQ Database Binding

This software represents a binding to objects and procedures that enable the Ada95[1] programmer to manipulate or query a relational database This document describes the design principles and goals of this APQ binding. It also supplies reference documentation to the programmer, enabling the reader to write applications using the PostgreSQL or MySQL databases, in the Ada programming language.

The APQ binding was initially developed using GNAT 3.13p under FreeBSD 4.4 release. APQ version 2.0 was developed using Debian Linux and GNAT 3.14p. The examples presented will be tested under the same development environment.

The source code avoids any use of GNAT specific language extensions. The possible exception to this rule is that the GNATPREP tool may be used to precompile optional support of optional databases. There is some C language source code used, to facilitate Ada and database C language library linkages. The following C language libraries are necessary in addition to the APQ client library, when linking your application:

Library	Database
libpq	PostgreSQL
libmysqlclient	MySQL

GNAT specific features are avoided where possible. The pragma:

```
pragma Linker_Options("-lapq");
```

is used for example, to save the programmer from having to specify linking arguments. Therefore those using non-ACT vendor supplied Ada compilers might be able to compile and use this binding without a huge investment.

A 32-bit Windows library for APQ can be built for use with the PostgreSQL and MySQL DLL client libraries. APQ release 2.1 should include the win32 build instructions necessary, but has been omitted in the first 2.0 release.

1.4.1 General Features

This binding supports all of the normal database functions that a programmer would want to use. Additionally blob support is included[2], and implemented using the Ada streams interface. This provides the programmer with the Ada convenience and safety of the streams interface when working with blobs.

This binding includes the following general features:

1. Open and Close one or more concurrent database connections
2. Create and Execute one or more concurrent SQL queries on a selected database connection

[1] Hereafter, we'll just refer to the language as Ada, even though the version of the language implied is Ada95.

[2] For PostgreSQL only, at release 2.0.

3. Begin work, Commit work or Rollback work

4. Access error message text

5. Generic functions and procedures to support specialized application types

6. The NULL indicator is supported

7. Blob support using the Ada streams facility

8. A wide range of native and builtin data types are supported

9. Database neutral API is now supported for most functions

1.4.2 Binding Type

This library represents a thick Ada binding to the PostgreSQL C programmer's libpq library [3], and with version 2.0, MySQL's C programming library. As a thick binding, there are consequently Ada objects and data types that are tailored specifically to the Ada programmer. Some data types and objects exist to mirror those used in the C language, while others are provided to make the binding easier or safer to apply.

A thin binding would have required the Ada programmer to be continually dealing with C language data type issues. Conversions to and from various types and pointers would be necessary making the use of the binding rather tedious. Furthermore, the resulting Ada program would be much harder to read and understand.

A thick binding introduces new objects and types in order to provide an API to the programmer. This approach however, fully insulates the Ada programmer from interfacing with C programs, pointers and strings. The design goal has additionally been to keep the number of new objects and types to a minimum. This has been done without sacrificing convenience and safety. Readability of the resulting Ada program was also considered to be important.

The objects and data types involved in the use of this binding can be classified into the following main groups:

1. Native data types and objects

2. Database manipulation objects

3. New database related objects and types for holding data

Native data types need no explanation in this document. The database manipulation objects will be described in section 1.6. The following section will introduce the Ada types that are used to hold data.

[3] C++ programs can also make use of this library but there exists the library libpq++ for C++ native support.

1.5 Binding Data Types

The PostgreSQL database supports many standard SQL data types as well as a few exotic ones. This section documents the database base types that are supported by the Ada binding to the database. This list is expected to grow with time as the Ada binding continues to mature in its own software development.

The "Data Type Name" column in the following table refers to a binding type if the type name is prefixed with "APQ_"[4]. These data types were designed to mimic common database data types in use. They can be used as they are provided, or you may subtype from them or even derive new types from them in typical Ada fashion. All other data types are references to native Ada data types (for some of these, the package where they are defined are shown in the "Notes" column).

The column labelled "Root Type" documents the data type that the APQ_ data type was derived from. Where they represent an Ada subtype, the column "Subtype" indicates a "Y". For type derivations a "N" is shown in this column, indicating that the APQ_ type listed is made "unique".

1.5.1 PostgreSQL Data Types

The "Notes" column of the table shows notes, package names and PostgreSQL data type names where the name is given in all capitals.

Data Type Name	Root Type	Subtype	PostgreSQL Notes
Row_ID_Type	-	N	Used for blobs and rows
String(<>)	-	-	Native Strings
String(a..b)	-	-	Native fixed length strings
Unbounded_String	-	-	Ada.Strings.Unbounded
Bounded_String	-	-	Ada.Strings.Bounded
APQ_Smallint	-	N	SMALLINT
APQ_Integer	-	N	INTEGER
APQ_Bigint	-	N	BIGINT
APQ_Real	-	N	REAL
APQ_Double	-	N	DOUBLE PRECISION
APQ_Serial	-	N	SERIAL
APQ_Bigserial	-	N	BIGSERIAL
APQ_Boolean	Boolean	Y	BOOLEAN
APQ_Date	Ada.Calendar.Time	Y	DATE
APQ_Time	Ada.Calendar.Day_Duration	Y	TIME (no timezone)
APQ_Timestamp	Ada.Calendar.Time	N	TIMESTAMP (no timezone)
APQ_Timezone	Integer	N	range -23..23
APQ_Bitstring	-	N	BIT or BIT VARYING
Decimal_Type	-	-	Package PostgreSQL.Decimal
range <>	-	-	Native Integers

[4] Formerly, the PostgreSQL specific types had used a PG_ prefix.

delta <>	-	-	Native Fixed Point
digits <>	-	-	Native Floating Point
delta <> digits <>	-	-	Native Decimal

The data type shown as “Decimal_Type” is special, in that it is supported from a child package APQ.PostgreSQL.Decimal. It represents a tagged type that provides an interface to the C routines used by the PostgreSQL database server, for arbitrary precision decimal values.

1.5.2 MySQL Data Types

The following table summarizes the MySQL specific data types and the corresponding APQ data types.

APQ Data Type	Ada Spec	Subtype	Comments
Row_ID_Type	unsigned 64 bits	N	For all databases
APQ_Smallint	signed 16 bits	N	SMALLINT
APQ_Integer	signed 32 bits	N	INTEGER
APQ_Bigint	signed 64 bits	N	BIGINT
APQ_Real	digits 6	N	REAL
APQ_Double	digits 15	N	DOUBLE [PRECISION]
APQ_Serial	range 1..2147483647	N	*INTEGER*
APQ_Bigserial	range 1..2**63	N	*BIGINT*
APQ_Boolean	Boolean	Y	BOOLEAN
APQ_Date	Ada.Calendar.Time	Y	DATE
APQ_Time	Ada.Calendar.Day_Duration	Y	TIME
APQ_Timestamp	Ada.Calendar.Time	N	TIMESTAMP
APQ_Timezone	range -23..23	N	*Not in MySQL*
APQ_Bitstring	array(Positive) of APQ_Boolean	N	*Not in MySQL*

Notice the italicized SQL keywords in the table. They identify the SQL keywords that differ from PostgreSQL. However, the programmer only needs to be concerned with these SQL keywords when creating new tables or temporary tables. For example a column of type SERIAL in a PostgreSQL table, should be declared as a INTEGER type in MySQL.

1.6 Database Objects

Much of the binding between Ada and the database server is provided through the use of tagged record types. Presently the APQ binding operates through the following three object types:

Root Type	Derived Type	Purpose	Notes	Finalized

Root_Connection_Type	Connection_Type	Connection	Required by queries and blobs	Yes
Root_Query_Type	Query_Type	SQL interface	Re-usable object.	Yes
N/A	Blob_Type	Blob interface	Must be in transaction	No

Note that the Connection_Type and Query_Type objects are automatically finalized when they go out of scope. The Blob_Type however, does not finalize automatically, because it represents an access type to a Blob_Object. This is similar in concept to an open a file, using the File_Type data type. This design approach was necessary in order to support the Streams oriented access to database blobs.

1.6.1 Object Hierarchy

Before multiple database products were supported, the APQ object hierarchy was simple. To provide generic level support however, there are now root objects and derived objects. In most application programming contexts, the writer does not need to be concerned with this fact. However, if you frequently inspect the spec files instead of the documentation, you must be aware that primitives for a given object may be declared in multiple places. Please examine the following chart:

Package Name	Description
APQ	Root objects and primitives
APQ.PostgreSQL	Declarations and constants unique to PostgreSQL
APQ.PostgreSQL.Client	Derived objects and added primitives
APQ.MySQL	Declarations and constants unique to MySQL
APQ.MySQL.Client	Derived objects and added primitives

From this chart, you can see that support for a given database is derived from the APQ level package. Root objects are declared in APQ, with common functionality. Some primitives must be overridden by the derived object. For example, APQ.Root_Query_Type declares a primitive named Value to return a string column result. If this particular method is called, the exception Is_Abstract will be raised, to indicate that it must be overriden with code to handle the specific database being used.

For this reason, the APQ.MySQL.Query_Type object for example, is derived from the APQ.Root_Query_Type object. This Query_Type object will provide its own implementation of the Value function to return a column result, and so will work as expected.

So when looking for primitives available to the Query_Type object, don't forget that many common primitives will be inherited from the APQ.Root_Query_Type object. The same is true for Connection_Type objects. They inherit a number of common primitives from the APQ.Root_Connection_Type object.

Chapter 2

Connecting to the Database

Before any useful work can be accomplished by a client program, a connection must be established between the Ada program and the database server. This chapter will demonstrate how to use the APQ binding to enable a program to connect and disconnect from the database server.

2.1 The Connection_Type

This object holds everything that is needed to maintain a connection to the database server. There are six groups of primitive operations for this object:

1. Context setting operations
2. Connection operations
3. Connection Information functions
4. General Information operations
5. Implicit operations (Finalization)
6. Trace Facilities
7. Generic Database Operations

2.2 Context Setting Operations

These primitives "configure" the connection that is to be made later. When the object is initially created, it is in the disconnected state. While disconnected, configuration changes can be made in to affect the next connection attempt. The application should not make configuration changes while the object is in the connected state.[1]

[1] This is probably not yet enforced by the current version of the APQ binding software.

The configuration primitives are the following[2]:

[2]The items marked “Root” are primitives from APQ.Root_Query_Type. The items marked “Derived” are those overrides that are declared on the APQ.*.Query_Type object.

Type	Derivation	Name	Purpose
proc	Root	Set_Host_Name	Set server host name
proc	Root	Set_Host_Address	Set server host IP address
proc	Root	Set_Port	Set server IP port number
proc	Root	Set_DB_Name	Set database name
proc	Root	Set_User_Password	Set userid and password
proc	Root	Set_Options	Set userid and password

2.2.1 PostgreSQL Defaults

The PostgreSQL database defines certain environment variables that can specify defaults. These and the fallback values are documented below:

Type	Derivation	Name	Default	Fallback
proc	Root	Set_Host_Name	PGHOST	localhost
proc	Root	Set_Host_Address	PGHOST	localhost
proc	Root	Set_Port	PGPORT	5432
proc	Root	Set_DB_Name	PGDATABASE	LOGNAME
proc	Root	Set_User_Password	PGUSER PGPASSWORD	LOGNAME
proc	Root	Set_Options	PGOPTIONS	""

The capitalized names shown in the "Default" and "Fallback" columns represent environment variable names. When any of the environment variables are undefined in the "Default" column, the value used is determined by the "Fallback" value listed. The fallback variable name LOGNAME is simply used to represent the current user's userid.[3] When no password value is provided and no PGPASSWORD environment variable exists, then no password is assumed.

2.2.2 Procedure Set_Host_Name

The Set_Host_Name procedure accepts the following arguments

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Host_Name	in		String	-

The following example configures the Connection_Type object to connect to host "witherspoon":

[3] The PostgreSQL libpq library may in fact, completely ignore the LOGNAME environment variable, and simply look up the userid in the /etc/password file.

```
declare
   C : Connection_Type;
begin
   Set_Host_Name(C,"witherspoon");
```

2.2.3 Procedure Set_Host_Address

The procedure takes two arguments, in the same fashion as Set_Host_Name:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Host_Address	in		String	-

The following example configures the Connection_Type object to connect to IP address 10.0.0.7:

```
declare
   C : Connection_Type;
begin
   Set_Host_Address(C,"10.0.0.7");
```

2.2.4 Procedure Set_Port

This procedure configures the port where the database server is listening (when using TCP/IP as the transport):

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Port_Number	in		Integer	-

The following code fragment shows how the port number is configured to use port 5432:

```
declare
   C : Connection_Type;
begin
   Set_Port(C,5432);
```

2.2.5 Procedure Set_DB_Name

This procedure call configures the name of the database that the server is to use when the connection is established:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	DB_Name	in		String	-

The following code fragment shows how the database name is configured to be "production":

```
declare
   C : Connection_Type;
begin
   Set_DB_Name(C,"production");
```

2.2.6 Procedure Set_User_Password

This procedure call configures both the userid and the password together. If there is no password, then supply the null string:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	User_Name	in		String	-
3	User_Password	in		String	-

The following example code fragment illustrates how the userid and password is configured:

```
declare
   C : Connection_Type;
begin
   Set_User_Password(C,"myuserid","xyzzy");
```

2.2.7 Procedure Set_Options

This procedure call permits the caller to specify any specialized database server options. The options are specified in string form with this API call. The specific options, and the format of those options will vary according to the database being used. See the following subsections for additional information about the database engine specifics.

The procedure **Set_Options** is documented as follows:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Options	in		String	-

The following PostgreSQL code fragment illustrates how two options may be configured:

```
declare
   C : Connection_Type;
begin
   Set_Options(C,"requiressl=1 dbname=test");
```

Note that in this example, the option string has been used to declare the database name to be used. Standard values should be set through the primitive functions provided.

Otherwise, when information primitives are added, you may not get correct results. Any non-standard options like the "requiressl" option, should be configured in this procedure call.

PostgreSQL Options

The documentation is not very clear about the format of these options, but it appears that keyword=value pairs separated by *spaces* for multiple options are accepted. If you must include spaces or other special characters within the value component, then you must follow PostgreSQL escaping rules. Refer to the database server documentation for these details.

MySQL Options

MySQL's C interface is much different than PostgreSQL's C interface for options. MySQL uses an enumerated value and argument pair when setting an option.[4] To keep the APQ interface friendly and consistent, APQ will accept all options and arguments in a string form as documented in section 2.2.7. However, these string options must be processed by APQ and digested into arguments usable by the MySQL C client interface. Consequently, APQ must anticipate these options and the option format in advance. For these reasons, the MySQL options and their arguments will be partially documented here.

The format of the option string should be one or more option names and arguments, separated by commas. Option names are treated as caseless (internally upcased).

```
Set_Options(C,"CONNECT_TIMEOUT=3,COMPRESS,LOCAL_INFIL=1");
```

Each option should be separated by a comma. APQ processes each option in left to right fashion, making multiple MySQL C API calls for each one.

The following is a list of APQ supported options:

Option Name	Argument Type	Comments
CONNECT_TIMEOUT	Unsigned	Seconds
COMPRESS	None	Compressed comm link
NAMED_PIPE	None	Windows: use a named pipe
INIT_COMMAND	String	Initialization command
READ_DEFAULT_FILE	String	See MySQL
READ_DEFAULT_GROUP	String	See MySQL
SET_CHARSET_DIR	String	See MySQL
SET_CHARSET_NAME	String	See MySQL
LOCAL_INFIL	Boolean	See MySQL

It is important to observe that any option that requires an argument, must have one. Any argument that requires an unsigned integer, must have an unsigned integer

[4] Although, some options do not use the argument.

(otherwise an exception is raised). A Boolean argument should be the value 0 or 1. At the present time, APQ gathers string data up until the next comma or the end of the string. Currently an option argument string cannot contain a comma character.[5]

2.2.8 Procedure Set_Notice_Proc

The PostgreSQL database[6] server sends notice messages back to the libpq C library, that the APQ binding uses. These are received by a callback, after certain database operations have been completed. While the messages are saved in the Connection_Type object (see also section 2.5.3), they overwrite each other as each new message comes in. For this reason, it may be desireable for some applications to also receive a callback, so that they can process the messages without losing them. The most common reason to do this is to simply display them on standard error.

The callback procedure must be defined as follows:

The default setting for any new Connection_Type object is No_Notify.

```
procedure Notice_Callback(C : in out Connection_Type; Message : String);
```

The Set_Notice_Proc takes an argument named Notify_Proc that is of the following type:

```
type Notify_Proc_Type is access
   procedure(C : in out Connection_Type; Message : String);
```

The Set_Notice_Proc procedure has the following calling signature:

Note that the Reset or Disconnect call will clear any registered Notify procedure.

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Notify_Proc	in		Notify_Proc_Type	-

This call can be made at any time to change the Notify procedure. The object may or may not be connected. The new procedure takes effect immediately upon return, and will be used when the object is connected. The present implementation only maintains one such procedure.[7]

Disabling Notify

The PostgreSQL.Client package provides the special constant No_Notify for the application programmer to use. An example of disabling notification follows:

```
declare
   C : Connection_Type;
begin
   ...
   -- Enable notify processing
```

[5] This needs to be corrected in a future release of APQ.
[6] The Set_Notice_Proc procedure is not available with MySQL.
[7] Note that the replaced procedure is not returned. A future implementation of APQ may address this.

```
Set_Notify_Proc(C,My_Notify'Access);
...
-- Disable notification
Set_Notify_Proc(C,No_Notify);
```

Using Standard_Error_Notify

During the debugging phase of a database application, it may be useful to simply have the notice messages printed on Standard_Error. To do this, simply provide the access constant *Standard_Error_Notify* as the second argument:

```
declare
   C : Connection_Type;
begin
   ...
   -- Send notices to stderr
   Set_Notify_Proc(C,Standard_Error_Notify);
   ...
```

2.3 Connection Operations

The APQ binding provides three primitives for connecting and disconnecting from the database server. They are summarized in the following table:

Type	Name	Purpose
proc	Connect	Connect to the database server
proc	Disconnect	Disconnect from the database server
proc	Reset	Disconnect if connected

2.3.1 Procedure Connect

This primitive initiates a connection attempt with the database server as configured by the section 2.2 primitives. If the connection succeeds, the procedure call returns.

The Connect primitive as of APQ 1.91 automatically executes a 'SET DATESTYLE TO ISO' command to guarantee that the APQ date routines will function correctly, even when the PGDATESTYLE environment variable may choose something other than ISO. This implies however, that APQ applications should always format date information in the ISO format.

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-

The following exceptions may occur:

Exception Name	Reason
Not_Connected	The connection attempt failed
Already_Connected	There is already a connection

The Already_Connected exception indicates that you need to disconnect first, or use another Connection_Type object if you are maintaining multiple connections.

The following is an example call:

```
declare
   C : Connection_Type;
begin
   ...
   begin
      Connect(C);
   exception
      when No_Connection =>
         ...; -- Handle connection failure
      when Already_Connected =>
         ...; -- Indicates program logic problem
      when others =>
         raise;
   end;
```

2.3.2 Connection Cloning

Application writers may want additional connections cloned from a given connection. A web server may want to do this for example. This could be performed by obtaining all of the connection information from the given connection and then proceed to configure a new connection, but this is tedious and error prone. To clone a new connection from an existing connection, simply use the Connect primitive with the following calling signature:

The trace settings of the Same_As object are not carried to the new object C. You must manually configure any trace settings you require in the newly connected object C.

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Same_As	in		Connection_Type'Class	-

This primitive configures *C* in the same way that connection *Same_As* is configured. Then it creates a connection to the database using these cloned parameters.

The following exceptions may occur:

Exception Name	Reason
Not_Connected	The connection attempt failed
Already_Connected	There is already a connection

The Not_Connected exception can be raised if the Same_As connection is not connected (it must be connected). This same exception can be raised if the new connection fails (this should rarely happen unless your database is suddenly taken down or a network failure occurs). The Already_Connected exception is raised if *C* is already connected.

The following example shows how a procedure *My_Subr* can clone a new connection:

```
procedure My_Subr(C : Connection_Type) is
```

```
   C2 : Connection_Type;
begin
   Connect(C2,C);  -- Clone a connection
```

2.3.3 Procedure Disconnect

The Disconnect primitive closes the connection that was previously established in the Connection_Type object. The Disconnect primitive uses the following arguments:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-

The following exceptions may occur:

Exception Name	Reason
No_Connection	There is no connection to disconnect

The following code fragment shows the procedure call in action:

```
declare
   C : Connection_Type;
begin
   ...
   begin
      Disconnect(C);
   exception
      when No_Connection =>
         ...; -- Indicates program logic problem
      when others =>
         raise;
   end;
```

2.3.4 Procedure Reset

The Reset primitive is provided so that the programmer can recycle the Connection_Type object for use in a subsequent connection. Without this primitive, the user would need to destroy the original and create a new Connection_Type. The Reset primitive accepts the following arguments:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-

In addition to closing the current connection, if it is open, the notification procedure is also deregistered (if there was a Set_Notify_Proc performed).

No exceptions should occur. If there is a connection pending, it is disconnected. If there is no connection pending, the call is ignored. The following shows an example of

its use:

declare

```
declare
  C : Connection_Type;
begin
   ...
   Reset(C);     -- C is now ready for re-use
```

2.4 Connection Information Operations

A modular piece of software may get handed a Connection_Type object as a parameter, and have a need to inquire about the details of the provided connection. The following function primitives return information about the connection:

Function Name	Information Returned
Host_Name	Host name of the connection
Port	Port Number or Port Pathname
DB_Name	Database name
User	User name for the database
Password	Password for the database
Options	Database option parameters

All of the functions (save one) have the following calling signature:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			String	

The Port primitive that returns a String is for use with database connections using a UNIX socket. The socket pathname is returned in this case. When used for TCP/IP connections, a numeric string representing the IP port number is returned.[8]

The Port function can return a String type as the rest of the functions do, or it can return an Integer type instead. This Port primitive has the following signature, and is useful when IP sockets are used:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			Integer	

When called on Connection_Type objects without a current connection, an empty string is returned for any value that has not been configured (for example if Set_Host_Name has not been called, Host_Name will return “”). If the value has been set, then that value

[8] A practical, although not foolproof test, is to look for a ’/’ character to see if it is a UNIX socket.

is returned as expected. Once the Connection_Type object is connected to the database however, the values will be values fetched from the library libpq instead.[9]

The following code sample shows how to extract the host name and database name for the current connection.

```
procedure My_Code(C : in out Connection_Type) is
   Host_Name :     String := Host_Name(C);   -- Get host name of database
   Database_Name : String := DB_Name(C);     -- Get database name
begin
   ...
```

2.5 General Information Operations

Due to the modular construction of software, it is sometimes necessary to query an object for its present state. The following primitives of the Connection_Type object are available for querying the state:

Type	Name	Purpose
func	Is_Connected	Indicates connected state
func	Error_Message	Returns a error message text

2.5.1 Function Is_Connected

The Is_Connected function returns a Boolean result that indicates the present state of the Connection_Type object. The arguments are as follows:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-

There are no exceptions raised by this primitive.

The following example shows how to test if the object C is currently supporting a connection. The example disconnects from the server, if it determines that C is connected.

```
declare
  C : Connection_Type;
begin
   ...
   if Is_Connected(C) then
      Disconnect(C);
      ...
```

[9] Normally, these values should agree with what was configured.

2.5.2 Function Error_Message

The Error_Message function makes it possible for the application to report why the connection failed. This information is often crucial to the user of a failed application. The arguments accepted are as follows:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			String	

There are no exceptions raised by this function. If there is no present connection or no present error to report, the null string is returned. The following example shows how the connection failure is reported:

```
with Ada.Text_IO;
...
declare
   use Ada.Text_IO;
   C : Connection_Type;
begin
   ...
   begin
      Connect(C);
   exception
      when No_Connection =>
        Put_Line(Standard_Error,"Connection Failed!");
        Put_Line(Standard_Error,Error_Message(C));
        ...
      when Already_Connected =>
         ...; -- Indicates program logic problem
      when others =>
         raise;
   end;
```

2.5.3 Function Notice_Message

The C libpq interface library[10] provides the APQ binding with certain notification messages during some calls, by means of a callback. Each time one of these notifications is received from the database server, the notification message is saved in the Connection_Type object (replacing any former notice message). The last notification message received can be retreived using the Notice_Message function:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			String	

[10]The Notice_Message function is not available for MySQL.

No exception is raised, and the null string is returned if no notice message has been registered.

The following example illustrates one example of the Notice_Message function:

```
with Ada.Text_IO;
...
declare
   use Ada.Text_IO;
   C : Connection_Type;
begin
   ...
   declare
      Msg : String := Notice_Message(C);
   begin
      if Msg'Length > 0 then
         Put_Line(Standard_Error,Msg);
         ...
```

2.5.4 In_Abort_State Function

Section 3.4 documents the Abort_State exception. This exception is raised in response to a status flag stored in the Connection_Type object. When a transaction is started, any SQL error will put the PostgreSQL database server into an "abort state", where all current and future commands will be ignored, for the connection[11]. To permit the application programmer to query this status, the In_Abort_State function can be used. It returns True, if an error has occurred within a transaction, which requires a Rollback_Work (section 3.4) call to clear this state. The calling requirements are summarized in the following table:

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			Boolean	True if in "abort state"

The following exceptions are possible:

Exception Name	Reason
Not_Connected	There is no connection to query

The following example shows how this function might be used:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   ...
```

[11] MySQL does not support this concept, and so it does not go into an abort state.

```
Begin_Work(Q,C);
...
Execute(Q,C);
...
if In_Abort_State(C) then
   Rollback_Work(Q,C);
   ...
end if;
```

2.6 Implicit Operations

There are a few implicit operations that are performed that the programmer should be aware of. They are:

- The Connection_Type is subject to Finalization
- A default Commit/Rollback operation can occur at Finalization

The programmer is encouraged to call Commit_Work or Rollback_Work explicitly, whenever possible. This way, the programmer is in complete control of the transaction outcome.

If a transaction has not been committed or rolled back, and the connected Connection_Type object is finalized[12], then the default action for commit or rollback occurs. The default for the APQ binding is to rollback the transaction, when the connection is still active. If the programmer has disconnected from the database prior to finalization, then no further action occurs. To change or control the default action, use the Set_Rollback_On_Finalize procedure described in the next section.

2.6.1 Set_Rollback_On_Finalize Procedure

The Set_Rollback_On_Finalize primitive allows the programmer to change the default action for the Connection_Type object. The calling requirements are summarized in the following table:

The primitive may be called at any time prior to the object's own finalization.

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
2	Rollback	in		Boolean	-

To change the default to COMMIT WORK when the Connection_Type object finalizes, peform the following call:

```
declare
   C : Connection_Type;
begin
   Set_Rollback_On_Finalize(C,False); -- Commit
```

[12] Usually because the Connection_Type object has fallen out of scope.

2.6.2 Will_Rollback_On_Finalize Function

Programs sometimes need to inquire about the state of the Connection_Type object that they may have been passed. To inquire about the commit or rollback default, the Will_Rollback_On_Finalize function can be called. The following table summarizes the calling requirements:

The primitive may be called at any time prior to the object's own finalization.

#	Argument	in	out	Type	Default
1	C	in		Connection_Type	-
	returns			Boolean	True if will ROLLBACK WORK

2.7 Trace Facilities

No matter how carefully a programmer writes a new program, problems develop that are often difficult to understand. With good tracing facilities the problem is not only easily understood, but it becomes easy to correct.

To gain trace support using APQ, it is only necessary to perform the following steps:

1. Open a trace capture file with Open_DB_Trace
2. Optionally enable/disable tracing at various points in the program with Set_Trace[13]
3. Perform your SQL operations
4. Close the trace capture file with Close_DB_Trace[14]

The Open_DB_Trace procedure takes a Trace_Mode_Type parameter that decides what trace content is being collected. The valid enumerated values are:

APQ.Trace_None Collect no trace information (no file is written/created)

APQ.Trace_DB Collect only C library trace information[15]

APQ.Trace_APQ Collect only APQ SQL trace information

APQ.Trace_Full Collect both database library (libpq) and Trace_APQ information

The Trace_None value is provided so that the Open_DB_Trace procedure does not need to be coded around if a trace variable is supplied, which may or may not request tracing. Close_DB_Trace can be called on a Connection_Type for which Trace_None is in effect, without any exception being thrown (the call is ignored).

Trace_DB provides only what the C library (libpq for PostgreSQL) provides. This may be useful to the database software maintainers, if they want a trace of the activity that you are reporting problems with.

[13] Tracing is enabled by default after a Open_DB_Trace call.
[14] Or allow it to be closed when the Connection_Type object is finalized.
[15] Prior to APQ 2.0, this was Trace_libpq.

Trace_APQ is what the author considers to be the most useful output format to an APQ developer. The trace output in this mode is such that the extra trace information is provided in SQL comment form. The actual queries that are executed are in their natural SQL form. The captured Trace_APQ trace then, is in a format that can be played back, reproducing exactly what the application performed.[16] The full trace or portions of it then can be used to help debug SQL related problems.

The following shows a sample of what the Trace_APQ output looks like:

```
-- Start of Trace, Mode = TRACE_APQ
-- SQL QUERY:
BEGIN WORK
;
-- Result: 'BEGIN'

-- SQL QUERY:
INSERT INTO DOCUMENT (NAME,DOCDATE,BLOBID,CREATED,MODIFIED,ACCESSED)
VALUES ('compile.adb','2002-08-12 21:09:25',3339004,'2002-08-12 21:59:48',
        '2002-08-12 21:09:25','2002-08-19 22:11:36')
;
-- Result: 'INSERT 3339005 1'

-- SQL QUERY:
SELECT DOCID
FROM DOCUMENT
WHERE OID = 3339005
;
-- Result: 'SELECT'
...
-- SQL QUERY:
COMMIT WORK
;
-- Result: 'COMMIT'
-- End of Trace.
```

The following subsections describe the primitives that provide support for trace facilities.

2.7.1 Procedure Open_DB_Trace

To start any capture of trace information, you must specify the name of the text file to be written to. The file must be writable to the current process. The Connection_Type object must be connected prior to calling Open_DB_Trace:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Filename	in		String	-
3	Mode	in		Trace_Mode_Type	Trace_APQ

The following exceptions are possible:

[16] There are limitations however, since the blob functions are not traced at the present release.

Exception Name	Reason
Not_Connected	There is no connection
Tracing_State	Trace is already enabled

Upon return from the Open_DB_Trace procedure, a text file will be created and ready to have trace entries written to it.[17]

The following example shows how a call might be coded:

```
declare
   C : Connection_Type;
begin
   ...
   Open_DB_Trace(C,"./bugs.sql",Trace_APQ);
```

2.7.2 Procedure Close_DB_Trace

Closing the tracing facility for a connection, suspends all further trace writes. Once this has been done, the effect of Set_Trace is superceeded, preventing any further trace information being written. The calling requirements are outlined in the following table:

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-

If the Open_DB_Trace call was made with the Mode parameter set to Trace_None, then the call to Close_DB_Trace has no effect and is ignored for programmer convenience.

No exceptions are raised.

An example call is shown below:

```
declare
  C : Connection_Type;
begin
   ...
   Open_DB_Trace(C,"./bugs.sql",Trace_APQ);
   ...
   Close_DB_Trace(C);
```

2.7.3 Procedure Set_Trace

In large applications where large numbers of SQL statements are executed, it may be desirable to trace only certain parts of its execution in a dynamic fashion. The Set_Trace primitive gives the programmer a way to disable and re-enable tracing at strategic points within the application. The calling requirements are summarized as follows:

[17] Note that trace entries are buffered by C standard I/O routines, so trace information may be held in memory buffers before it is flushed out or closed.

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
2	Trace_On	in		Boolean	True

Tracing is enabled by default, after a successful call to Open_DB_Trace is made (unless Mode was Trace_None).

There are no exceptions raised.

Note that it is considered safe to invoke Set_Trace, even if a former Open_DB_Trace call was not successfully performed, or the trace mode was Trace_None. This allows the application to retain strategic Set_Trace calls without having to remove them, when the Open_DB_Trace call is disabled[18] or commented out.

2.7.4 Function Is_Trace

It may be helpful to the developer that is tracking down a problem to know when tracing is enabled or not. The Is_Trace function returns true when the trace collection file is receiving trace information. The calling arguments are listed below:

The returned value tracks the last value set by Set_Trace. True can be returned even when a trace file is not open when Trace_None is used, or no Open_DB_Trace was called.

#	Argument	in	out	Type	Default
1	C	in	out	Connection_Type	-
	returns			Boolean	

Note that the initial state of the Connection_Type object is to have Is_Trace to return True. Also after a successful Open_DB_Trace, Is_Trace will return True.

An example showing its use is given below:

```
declare
  C : Connection_Type;
begin
   ...
   Open_DB_Trace(C,"./bugs.sql",Trace_APQ);
   ...
   if Is_Trace(C) then
      -- We are collecting trace info
```

2.8 Generic Database Operations

APQ 2.0 is designed so that all but the most specialized database operations, can be performed, given only a Root_Connection_Type'Class object (declared in top level package APQ). The following sections describe some generic database related primitives that are necessary for successful generic database support.

[18] Setting Mode to Trace_None effectively disables the trace facility without requiring any code changes.

2.8.1 Package APQ

Root object support is provided in the package APQ. Generic database code will normally only use this package:

```
with APQ;
use APQ;    -- Optional use clause
```

The data types that will be used will be:

- APQ.Root_Connection_Type
- APQ.Root_Query_Type

The generic primitives that will be covered in the next section are:

- APQ.Engine_Of
- APQ.New_Query

2.8.2 Predicate Engine_Of

Given a Root_Connection_Type'Class object, generic database code sometimes needs to determine which specific database is being used. This allows the code to make special SQL syntax changes, depending upon the technology being used (for example, MySQL permits the use of a LIMIT keyword in queries).

The **Engine_Of** primitive (dispatching) will identify the database technology that is being used:

#	Argument	in	out	Type	Default	Description
1	C	in		Root_Connection_Type	-	The connection object
returns				Database_Type		The database engine used

The data type Database_Type is currently defined as follows (more database engines may follow in future APQ releases):

```
type Database_Type is (
   Engine_PostgreSQL,
   Engine_MySQL
);
```

The following example code shows how to test if a PostgreSQL database is being used:

```
with APQ; use APQ;
...
procedure App(C : Root_Connection_Type'Class) is
begin
   ...
   if Engine_Of(C) = Engine_PostgreSQL then
      ...
```

2.8.3 Primitive New_Query

Normally, an application database procedure will receive a connection object as one of its input parameters. Generally, this connection is established in the main program and then used by the program components as required. However, to pass the parameter in a generic way (allowing for polymorphism), you would declare the procedure's argument as receiving data type Root_Connection_Type'Class.

Within the called procedure however, you will need a Query_Type object. This too could be passed in as an argument, but this is unnecessary. What you need is a convenient way to create a Query_Type object that matches the connection that you have received as a parameter. In other words, if your connection object is a:

```
APQ.PostgreSQL.Client.Connection_Type
```

object, then your application will want to create a:

```
APQ.PostgreSQL.Client.Query_Type
```

object. You want to avoid tests like:

```
if Connection is in APQ.PostgreSQL.Client.Connection_Type then
   ...
elsif Connection is in APQ.MySQL.Client.Connection_Type then
   ...
```

The above type of code would force your generic code to also with the packages:

```
with APQ.PostgreSQL.Client;
with APQ.MySQL.Client;
etc.
```

which would be very inconvenient and unnecessary.

To make generic code easier, APQ provides a dispatching Query_Type object factory primitive that can be used for this purpose. For example:

```
with APQ;
use APQ;
procedure My_Generic_App(C : Root_Connection_Type'Class) is
   Q : Root_Query_Type'Class := New_Query(C);
begin
   Prepare(Q,"SELECT NAME, INCOME");
   Append_Line(Q,"FROM SALARIES");
```

The assignment line (for Q) shows the application of the primitive **New_Query**. This dispatching primitive returns the correct Query_Type object that matches the connection that was given. The primitive **New_Query** is more formally presented as follows:

#	Argument	in	out	Type	Default	Description
1	C	in		Root_Connection_Type	-	The SQL connection object
returns				Root_Query_Type'Class		The new Query_Type object

2.8.4 Query_Type Assignment

Prior to APQ version 2.0, the Query_Type object was a *limited* tagged type. This meant that the Query_Type object was never able to be assigned to another Query_Type object. With the need for a factory primitive like **New_Query** it was necessary to lift that restriction (otherwise the factory was unable to return the created object). So the Root_Query_Type and derived forms, permit assignment as of APQ 2.0 and later.

When a Query_Type is assigned in APQ, nothing spectacular happens. In fact, the contents of the object on the right hand side are effectively ignored, leaving a new object on the left side. The following example shows how Q1 and Q2 are essentially the same:

```
declare
   Q0 : Query_Type;
   Q1 : Query_Type;
   Q2 : Query_Type;
begin
   ...
   Q1 := Q0;   -- Q1 becomes initialized (Q0 ignored)
   Clear(Q2);  -- Initialize Q2
```

In this example, both Q1 and Q2 end up in the same state, and no state information is taken from Q0. You might be wondering why would you implement such a thing? The following generic example illustrates why this is convenient and useful:

```
with APQ;
use APQ;
procedure My_Generic_App(C : Root_Connection_Type'Class) is
   Q : Root_Query_Type'Class := New_Query(C);
begin
   Prepare(Q,"SELECT NAME, INCOME");
   Append(Q,"FROM SALARIES");
   Execute(Q,C);
   ...
   declare
      Q2 : Root_Query_Type'Class := Q;
   begin
      ...
```

The example illustrates that the assignment is simply a convenient factory of its own kind. It is also likely to be slightly more efficient than the **New_Query** primitive on the connection. Think of assignment of Query_Type objects as cloning operations. The assigned object becomes a fresh initialized clone of the Query_Type object on the right hand side of the assignment.

If you are still scratching your head about this, consider a concrete example:

1. My_Generic_App is called with an argument of type APQ.MySQL.Client.Connection_Type
2. Q gets assigned a new object of type APQ.MySQL.Client.Query_Type to match the connection (it is a MySQL connection).
3. Q2 gets assigned a new object of type APQ.MySQL.Client.Query_Type to match the connection.

If the procedure My_Generic_App is called with a PostgreSQL connection, then PostgreSQL Query_Type objects will be ussed in the procedure instead. This is polymorphism at work.

Chapter 3

SQL Query Support

Once a database connection has been established, the application is ready to invoke operations on the database server. To ease the programmer's burden in keeping track of the various components involved in these transactions, the Query_Type object is provided. The Query_Type object and the Connection_Type object are often used together. Some primitives do not involve the connection, while others do.

There are a large number of primitives associated with the Query_Type object. Most of them are related to the large number of data types that are supported. These primitives fall into the following basic categories:

1. Object initialization
2. SQL Query building
3. SQL Execution
4. Transaction operations
5. Fetch operations
6. Column information functions
7. Value fetching functions
8. Value and Indicator fetching procedures
9. Information operations

In addition to these, are a number of generic functions and procedures that permit the APQ user to custom tailor the API to his own specialized Ada data types.

3.1 Initialization

The Query_Type object is initialized when the object is instantiated. However, the Query_Type object is very often re-used as various SQL operations are performed by a program. To re-use the Query_Type object, one of the following two calls may be used to recycle it for re-use:

Type	Name	Purpose
proc	Clear	Clear object and re-initialize
proc	Prepare	Reinitialize with start of new SQL query

The Clear procedure does the initialization of the Query_Type object. The Prepare primitive also invokes Clear.[1] The Prepare primitive additionally starts the building of an SQL query. For short SQL statements, may comletely specify the entire SQL statement.

3.1.1 Procedure Clear

The Clear primitive completely resets the state of the Query_Type object and accepts the following arguments:

#	Argument	in	out	Type	Default
1	Q	in	out	Query_Type	-

There are no exceptions raised by this call.

The use of the Clear primitive is recommended after all SQL processing related to the query has been completed. This permits any database server results to be released. Think of it as “closing” the query.

The following example illustrates it’s use:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   ...
   Clear(Q);
```

3.1.2 Procedure Prepare

The Prepare primitive goes one step further than Clear in that it readies the object for the start of an SQL statement build. If the query is short, this will be the only building step required. The Prepare procedure takes the following arguments:

#	Argument	in	out	Type	Default	Description

[1] Consequently, your application need not invoke Clear() prior to calling Prepare().

1	Q	in	out	Query_Type	-	
2	SQL	in		String		Starting SQL text
3	After	in		String	Line_Feed	Append to SQL text

The *SQL* argument defines the start of your SQL statement. The *After* argument may supply either the default (line feed) or some other text to append to the SQL text.[2] It is provided as a programmer convenience, since many times the programmer will need to append a comma, for example.

There are no exceptions raised by this call.

The following code shows an example of building a query to drop a table:

```
declare
   Q : Query_Type;
begin
   ...
   Prepare(Q,"DROP TABLE DRONE");
```

3.2 SQL Query Building

The previous section primitives "cleared" the Query_Type for a new query. The primitives provided in this section help to build a new SQL query or to continue (append to) the one started by the Prepare call in section 3.1.2. The programmer may start with a Prepare call and follow it by a number of "append" calls[3], or he may call Clear and build upon an empty query and skip the use of Prepare instead.

There are two broad categories of support for creating SQL queries. They are:

1. Append a value to the SQL query
2. Encode a value or NULL, to the SQL query.

Both of these categories append to the current query. Primitives in category 2 , are prefixed with *Encode* and will be described later in the present chapter.

The Append category of support is useful for values that are never NULL (in SQL terms these columns that are declared as "NOT NULL"). The Encode category of support is provided for values in your application that may be in the NULL state. It is not absolutely required that the Encode support be used, since it is possible for the application to test for a NULL value. However, the programmer will find that the Encode support provides application coding convenience and economy of expression. With compact code, better readability and safety is obtained.

Within category 1, there are five groups of primitives[4] that build on the present query. They are:

1. Append a string

[2] Don't forget to allow for additional blanks and commas and such.

[3] This is probably preferred, since the Prepare call tends to be a good marker for the start of a query.

[4] The generic procedures have been lumped in with the primitives.

2. Append a string and a *"newline"*
3. Append a quoted string
4. Append non string types
5. Append using generic procedures for custom types

Encode support on the other hand, only provides for the needs of variables that must be communicated to the database server. As a result, the encode procedures consist only of the following two groups:

1. Encode non-string types
2. Encode using generic procedures for custom types

Presently only the second group is provided for by the APQ binding.[5] A future release may provide builtin support where there currently exists Append support in group 1.

The append procedures (category 1) will be described first and then followed by the encode procedures (category 2).

3.2.1 Append SQL String

There are two Append procedures for adding SQL text to the Query_Type object. The difference between them is only in the data type of the SQL argument (#2):

#	Argument	in out	Type	Default	Description
1	Q	in out	Query_Type	-	
2	SQL	in	String Ada.Strings.Unbounded.Unbounded_String	-	SQL text to append
3	After	in	String	""	Append to SQL text

There are no exceptions raised by this call.

The following example shows how Append is used:

```
declare
   Q : Query_Type;
begin
   ...
   Prepare(Q,"SELECT CUSTNO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
```

Note that the Prepare call implies a default argument After=New_Line. The calls to Append merely append the text that you provide to continue the current line. If you want to put a line feed at the end of "FROM CUSTOMER", you can either supply the

[5] The reasoning is that most of the time, the user will want to instantiate the generic procedures anyway. This permits both the data type and the null indicator type to be a custom application type.

string New_Line to argument “After” in the Append call, or you can call Append_Line (See section 3.2.2), which is perhaps clearer code to read.

The After argument is designed to make it easier to build queries because often commas are required between items. The following example illustrates:

```
declare
   Q :          Query_Type;
   Col_Name_1 : String := “CUSTNO”;
   Col_Name_2 : String := “CUST_NAME”;
begin
   ...
   Prepare(Q,”SELECT ”);
   Append(Q,Col_Name_1,”,”);
   Append(Q,Col_Name_2,APQ.Line_Feed);
   Append(Q,”FROM CUSTOMER”);
```

This example builds up the same query as the previous example did, except that the column names were provided by string variables.

3.2.2 Append SQL Line

The Append_Line procedure is provided for added convenience and program readability. The same effect can be had with a string Append call, using string APQ.Line_Feed supplied as the *After* argument. The Append_Line procedure has the following arguments:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	SQL	in		String		SQL text

The Append_Line procedure is one of the few that does not sport an *After* argument.

3.2.3 Append Quoted SQL String

The Append_Quoted procedure call is designed to make it easier for the programmer to supply a string value that may contain special characters within it. Since a string value is already supplied with outer single quotes, any single quote appearing within the string must be quoted. The Append_Quoted procedure provides the necessary outer quotes for the SQL query, and escapes any special characters occuring in the string as well. The two Append_Quoted procedure calls differ only in the data type of the *SQL* argument:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	

2	SQL	in		String Ada.Strings.Unbounded.Unbounded_String	-	SQL text to quote
3	After	in		String	""	Additional SQL text

The following example illustrates the use of this call (using the String type):

```
declare
   Q :              Query_Type;
   Freds_Emporium : String := "Fred's Emporium";
begin
   ...
   Prepare(Q,"SELECT COMPNO,COMPANY_NAME");
   Append_Line(Q,"FROM SUPPLIER");
   Append(Q,"WHERE COMPANY_NAME = ");
   Append_Quoted(Q,Freds_Emporium,New_Line);
```

The effect of these calls is to build an SQL query that looks as follows:

```
SELECT COMPNO,COMPANY_NAME
FROM SUPPLIER
WHERE COMPANY_NAME = 'Fred\'s Emporium'
```

Notice how the quote character was escaped for use by the database server.

3.2.4 Append Non String Types to SQL Query

A fairly large set of builtin APQ data types are supported by varied Append calls that differ in the second argument *V*. The calling requirements can be summarized in the following table:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	V	in		Boolean APQ_Date APQ_Time APQ_Timestamp APQ_Bitstring Row_ID_Type		SQL value to convert into text
3	After	in		String	""	Additional SQL text

These Append procedure calls automatically convert the supplied data type in argument *V* into a string using a To_String function appropriate to the data type. Internall, the string Append procedure is then utilized to perform the remaining work. The following example shows how to apply these Append procedure calls:

```
declare
   Q :          Query_Type;
```

```
      Ship_Date :  APQ_Date;
   begin
      ...
      Prepare(Q,"SELECT COMPNO,COMPANY_NAME,SHIP_DATE");
      Append_Line(Q,"FROM SUPPLIER");
      Append(Q,"WHERE SHIP_DATE = ");
      Append(Q,Ship_Date,New_Line);
```

The example presented builds an SQL query that looks like this:

```
SELECT COMPNO,COMPANY_NAME,SHIP_DATE
FROM SUPPLIER
WHERE SHIP_DATE = '2002-07-21'
```

Notice that the Append call for APQ_Date automatically supplies the necessary quotes to the SQL query. All of the data types supported are moulded into a format that is acceptable in SQL syntax.[6]

There is one additional Append procedure call that has a special set of arguments in order to support dates with time zones. The arguments for this procedure call are as follows:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	TS	in		APQ_Timestamp	-	Date & Time
3	TZ	in		APQ_Timezone	-	Time zone
4	After	in		String	""	Additional SQL Text

Apart from the different argument names *TS* and *TZ*, this procedure works in the same fashion as the former Append procedure call. The *TZ* argument simply supplies the additional time zone information to be added to the timestamp.

3.2.5 Generic Append SQL Procedures

Ada programmers often take advantage of the strong typing that is available in the language. To accomodate this programming aspect, generic procedures are available so that type conversions are unnecessary. The following table documents the generic procedures that accept one generic argument named *Val_Type* and the data types that they support:

Procedure Name	Data Type	Notes
Append_Boolean	is new Boolean	Any Boolean type
Append_Integer	is range <>	Any signed integer type
Append_Modular	is mod <>	Any modular type
Append_Float	is digits <>	Any floating point type
Append_Fixed	is delta <>	Fixed point types

[6] Some of these formats may be database specific.

Append_Decimal	is delta <> digits <>	Any decimal type
Append_Date	is new Ada.Calendar.Time	Any date
Append_Time	is new Ada.Calendar.Day_Duration	Any time
Append_Timestamp	is new APQ_Timestamp	Time stamps
Append_Bitstring	is new APQ_Bitstring	Bit strings

Each of the resulting instantiated procedures provide the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	V	in		Val_Type	-	To be converted into SQL text
3	After	in		String	""	Additional SQL text

The following documents how these are instantiated and used:

```
declare
   type Price_Type is delta 0.01 digits 12;
   procedure Append is new Append_Decimal(Price_Type);
   Q :              Query_Type;
   Selling_Price :  Price_Type;
begin
   ...
   Prepare(Q,"UPDATE SUPPL_ORDER");
   Append(Q."SET SELLING_PRICE = ");
   Append(Q,Selling_Price,New_Line);
   Append_Line(Q,"WHERE ...");
```

In this example, the application defines its own unique type *Price_Type*. After instantiating the Append_Decimal generic procedure as Append, the application is free to neatly append a price value in Selling_Price, as if it were natively supported.

3.2.6 Generic Append_Timezone

The Append_Timezone has an additional generic paramter, and the instantiated procedure has a slightly different set of calling arguments. The generic parameters are documented as follows:

Argument Name	Data Type	Notes
Date_Type	is new Ada.Calendar.Time	Any date type
Zone_Type	is new APQ_Timezone	Any type derived from APQ_Timezone

The instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description

1	Q	in	out	Query_Type	-	
2	V	in		Date_Type	-	Date To be converted into SQL text
3	Z	in		Zone_Type	-	Time zone value
4	After	in		String	“”	Any additional SQL text

The following shows an example of its use:

```
declare
   type Ship_Date_Type is new APQ_Timestamp;
   type Ship_Zone_Type is new APQ_Timezone;
   procedure Append is new Append_Timezone(Ship_Date_Type,Ship_Zone_Type);
   Q :              Query_Type;
   Ship_Date :      Ship_Date_Type;
   Ship_Zone :      Ship_Zone_Type;
begin
   ...
   Prepare(Q,”SELECT COUNT(*)”);
   Append_Line(Q,”FROM ORDER”);
   Append(Q,”WHERE SHIP_DATE = “);
   Append(Q,Ship_Date,Ship_Zone,New_Line);
   ...
```

The example shows how the application’s types *Ship_Date_Type* and *Ship_Zone_Type* are accomodated by the Append instantiation of the generic procedure.

3.2.7 Generic Append of Bounded SQL Text

To accomodate the use of the package Ada.Strings.Bounded, the generic procedure Append_Bounded was provided. Its instantiation requirements differ from the preceeding ones because the instantiation of the Bounded_String type must be provided to the Append_Bounded generic procedure. The generic procedure is defined as follows:

```
generic
   with package P is new Ada.Strings.Bounded.Generic_Bounded_Length(<>);
procedure Append_Bounded(
   Q :     in out Query_Type;
   SQL :   in     P.Bounded_String;
   After : in     String);
```

In other words, Append_Bounded can be instantiated from any instantiation of the Ada.Strings.Bounded.Generic_Bounded_Length package. The example makes this easier to understand:

```
with Ada.Strings.Bounded;
...
declare
   package B80 is new Ada.Strings.Bounded.Generic_Bounded_Length(80);
   package B20 is new Ada.Strings.Bounded.Generic_Bounded_Length(20);
   procedure Append is new Append_Bounded(B80);
   procedure Append is new Append_Bounded(B20);
   Q :              Query_Type;
   Item_Code :      B20;
```

```
   Item_Name :      B80;
begin
   ...
   Prepare(Q,"SELECT COUNT(*)");
   Append_Line(Q,"FROM ORDER");
   Append(Q,"WHERE ITEM_CODE = ',""');
   Append(Q,Item_Code,"' AND ITEM_NAME = '");
   Append(Q,Item_Name,""' & New_Line);
   ...
```

The example shows how two different generic procedures named Append are instantiated from the Bounded_String instantiations *B80* and *B20*. Note that the Append_Bounded procedure does not escape special characters, nor provide the outer quotes.

3.2.8 Generic Append_Bounded_Quoted Procedure

To accomodate the quoting needs of Bounded_Strings, the Append_Bounded_Quoted generic procedure may be used:

```
generic
   with package P is new Ada.Strings.Bounded.Generic_Bounded_Length(<>);
procedure Append_Bounded_Quoted(
   Q :     in out Query_Type;
   SQL :   in     P.Bounded_String;
   After : in     String);
```

It is otherwise very similar to the previous Append_Bounded procedure. The following example illustrates a safer version of the prior example:

```
with Ada.Strings.Bounded;
...
declare
   package B80 is new Ada.Strings.Bounded.Generic_Bounded_Length(80);
   package B20 is new Ada.Strings.Bounded.Generic_Bounded_Length(20);
   procedure Append_Quoted is new Append_Bounded_Quoted(B80);
   procedure Append_Quoted is new Append_Bounded_Quoted(B20);
   Q :              Query_Type;
   Item_Code :      B20;
   Item_Name :      B80;
begin
   ...
   Prepare(Q,"SELECT COUNT(*)");
   Append_Line(Q,"FROM ORDER");
   Append(Q,"WHERE ITEM_CODE = ");
   Append_Quoted(Q,Item_Code," AND ITEM_NAME = ");
   Append_Quoted(Q,Item_Name,New_Line);
   ...
```

The instantiations of Append_Quoted[7] here will properly escape any special characters that may appear in the program's string variables Item_Code and Item_Name. Additionally, note that the outer quotes are provided automatically, easing the programmer's burden in building up the SQL query.

[7] It is not necessary to instantiate these procedures as *Append_Quoted*, but it is recommended for readability.

3.2.9 Encoding Quoted Strings

While strings are well covered by the category 1 support, it is necessary to encode a NULL in place of a quoted string, if the value's indicator indicates that the value is null. The instantiation arguments are as follows for Encode_String_Quoted:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean indicator type

The instantiation of Encode_String_Quoted has the following procedure arguments:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	SQL	in		String	-	String data value
3	Indicator	in		Ind_Type	-	NULL Indicator
4	After	in		String	""	Any additional SQL text

An example of its instantiation and use is shown below:

```
declare
   type Cust_Name_Ind_Type is new Boolean;
   procedure Encode_Quoted is new Encode_String_Quoted(Cust_Name_Ind_Type);
   Q :              Query_Type;
   Cust_Name :      String(1..30);
   Cust_Name_Ind :  Cust_Name_Ind_Type;   -- NULL Indicator for Cust_Name
begin
   ...
   Prepare(Q,"UPDATE CUSTOMER");
   Append_Line(Q,"SET CUST_NAME = ");
   Encode_Quoted(Q,Cust_Name,Cust_Name_Ind);
```

In this example, the String *Cust_Name* is given outer quotes and any special characters are escaped before the value is appended to the current SQL query being collected in object *Q*. If however, the indicator *Cust_Name_Ind* is True (indicating that the value *Cust_Name* should be interpreted as NULL), then the string "NULL" is appended instead. When NULL is supplied, no outer quotes are supplied. The following two SQL statements are possible, depending upon *Cust_Name_Ind*. When the indicator is false, a quoted value is supplied:

```
UPDATE CUSTOMER
SET CUST_NAME = 'Fred Willard'
...
```

When the indicator is true, the resulting query becomes this instead:

```
UPDATE CUSTOMER
SET CUST_NAME = NULL
...
```

3.2.10 Encoding Quoted Unbounded_String

To provide quoting support for Unbounded_Strings, the Encode_Bounded_Quoted generic procedure has been supplied. The generic procedure only requires one generic argument:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean indicator type

After Encode_Bounded_Quoted has been instantiated, the resulting procedure has the following signature:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	SQL	in		Ada.Strings.Unbounded.Unbounded_String	-	String data value
3	Indicator	in		Ind_Type	-	NULL Indicator
4	After	in		String	""	Any additional SQL text

An example of its use is as follows:

```
declare
   use Ada.Strings.Bounded;
   type Cust_Name_Ind_Type is new Boolean;
   procedure Encode_Quoted is new Encode_Unbounded_Quoted(Cust_Name_Ind_Type);
   Q :              Query_Type;
   Cust_Name :      Unbounded_String;
   Cust_Name_Ind :  Cust_Name_Ind_Type;   -- NULL Indicator for Cust_Name
begin
   ...
   Prepare(Q,"UPDATE CUSTOMER");
   Append_Line(Q,"SET CUST_NAME = ");
   Encode_Quoted(Q,Cust_Name,Cust_Name_Ind);
```

In this example, the Unbounded_String *Cust_Name* is given outer quotes and any special characters are escaped before the value is appended to the current SQL query being collected in object *Q*. If however, the indicator *Cust_Name_Ind* is True (indicating that the value *Cust_Name* should be interpreted as NULL), then the string "NULL" is appended instead. When NULL is supplied, no outer quotes are supplied. The following two SQL statements are possible, depending upon *Cust_Name_Ind*. When the indicator is false, a quoted value is supplied:

```
UPDATE CUSTOMER
SET CUST_NAME = 'Fred Willard'
...
```

When the indicator is true, the resulting query becomes this instead:

```
UPDATE CUSTOMER
SET CUST_NAME = NULL
...
```

3.2.11 Encoding Bounded Quoted Strings

Bounded strings require instantiations of Ada.Strings.Bounded.Generic_Bounded_Length with a specific length. This means that the instantiation of the package must be provided as an argment to the Encode_Bounded_Quoted instantiation:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean indicator type
P	Ada.Strings.Bounded.Generic_Bounded_Length(<>)	Package instantiation

The instantiated procedure has the following signature:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	SQL	in		P.Bounded_String	-	String data value
3	Indicator	in		Ind_Type	-	NULL Indicator
4	After	in		String	""	Any additional SQL text

An example showing its use is given below:

```
with Ada.Strings.Bounded;
declare
   package B80 is new Ada.Strings.Bounded.Generic_Bounded_Length(80);
   type Cust_Name_Ind_Type is new Boolean;
   procedure Encode_Quoted is new
      Encode_Bounded_Quoted(Cust_Name_Ind_Type,B80);
   Q :              Query_Type;
   Cust_Name :      B80.Bounded_String;
   Cust_Name_Ind :  Cust_Name_Ind_Type;   -- NULL Indicator for Cust_Name
begin
   ...
   Prepare(Q,"UPDATE CUSTOMER");
   Append_Line(Q,"SET CUST_NAME = ");
   Encode_Quoted(Q,Cust_Name,Cust_Name_Ind);
  ...
```

In this example, the Bounded_String *Cust_Name* is given outer quotes and any special characters are escaped before the value is appended to the current SQL query being collected in object *Q*. If however, the indicator *Cust_Name_Ind* is True (indicating that the value *Cust_Name* should be interpreted as NULL), then the string "NULL" is appended instead. When NULL is supplied, no outer quotes are supplied. The following two SQL statements are possible, depending upon Cust_Name_Ind. When the indicator is false, a quoted value is supplied:

```
UPDATE CUSTOMER
SET CUST_NAME = 'Fred Willard'
...
```

When the indicator is true, the resulting query becomes this instead:

```
UPDATE CUSTOMER
SET CUST_NAME = NULL
...
```

3.2.12 Encoding Non String Values

There are a large number of generic encoding procedures that follow the same general formula. The generic procedures are summarized in the following table:

Procedure Name	Data Type	Notes
Encode_Boolean	is new Boolean	Any Boolean type
Encode_Integer	is range <>	Any signed integer type
Encode_Modular	is mod <>	Any modular type
Encode_Float	is digits <>	Any floating point type
Encode_Fixed	is delta <>	Fixed point types
Encode_Decimal	is delta <> digits <>	Any decimal type
Encode_Date	is new Ada.Calendar.Time	Any date
Encode_Time	is new Ada.Calendar.Day_Duration	Any time
Encode_Timestamp	is new APQ_Timestamp	Time stamps
Encode_Bitstring	is new APQ_Bitstring	Bit strings

Instantiation of these generic procedures, require the following two parameters:

Argument Name	Data Type	Notes
Val_Type	<>	Type must correspond to generic procedure name
Ind_Type	is new Boolean	Any Boolean indicator type

All of these generic procedures instantiate a procedure with the following call signature:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	V	in		Val_Type	-	SQL data value to encode
3	Indicator	in		Ind_Type	-	NULL Indicator
4	After	in		String	“”	Any additional SQL text

The following example shows the use of one of these generic procedures:

```
declare
   type Cust_No_Type is new Integer range 1000..100_000;
   type Cust_Age_Type is new Integer range 0..200;
   procedure Append is new Append_Integer(Cust_No_Type);
   procedure Encode is new Encode_Integer(Cust_No_Type,Boolean);
   Q :            Query_Type;
   Cust_No :      Cust_No_Type;  -- Customer # NOT NULL
   Cust_Age :     Cust_Age_Type; -- Customer's age (can be NULL)
   Cust_Age_Ind : Boolean;       -- NULL Indicator for Cust_Age
```

```
   Cust_Name :    String(1..30); -- Customer Name NOT NULL
begin
   ...
   Prepare(Q,"INSERT INTO CUSTOMER (CUST_NO,AGE,CUST_NAME)");
   Append(Q,"VALUES (");
   Append(Q,Cust_No,",");
   Encode(Q,Cust_Age,Cust_Age_Ind,",");
   Append_Quoted(Q,Cust_Name,")" & New_Line);
   ...
```

Assuming the *Cust_Name* variable holds "Martin Mull", and *Cust_No* holds 12345, two possible SQL queries are possible, depending upon the value of *Cust_Age_Ind*, the null indicator. When *Cust_Age_Ind* is false (not null), then the SQL query would be:

```
INSERT INTO CUSTOMER (CUST_NO,AGE,CUST_NAME)
VALUES (12345,52,'Martin Mull')
```

When the indicator *Cust_Age_Ind* is true (representing null), then the query would be constructed as follows:

```
INSERT INTO CUSTOMER (CUST_NO,AGE,CUST_NAME)
VALUES (12345,NULL,'Martin Mull')
```

Notice how Append procedures are used for values that can never be null (no null indicator is involved). Encode routines are only necessary when a null indicator may require the encoding of the value NULL.

3.2.13 Encoding Timezone

Encoding APQ_Timezone values requires a special generic procedure named Encode_Timezone. Its generic parameters are described by the following table:

Argument Name	Data Type	Notes
Date_Type	is new APQ_Timestamp	Any APQ_Timestamp derived type
Zone_Type	is new APQ_Timezone	Any APQ_Timezone derived type
Ind_Type	is new Boolean	Any Boolean indicator type

The instantiated procedure has the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Query_Type	-	
2	D	in		Date_Type	-	Date value to encode
3	Z	in		Zone_Type	-	Time zone to encode
4	Indicator	in		Ind_Type	-	NULL Indicator
5	After	in		String	""	Any additional SQL text

The following example demonstrates the instantiation and use of the procedure:

```
declare
   type Cust_No_Type is new Integer range 1000..100_000;
   type Birthday_Type is new APQ_Timestamp;
   procedure Append is new Append_Integer(Cust_No_Type);
   procedure Encode is new Encode_Integer(Cust_No_Type,Boolean);
   procedure Encode is new
      Encode_Timezone(Birthday_Type,APQ_Timezone,Boolean);
   Q :            Query_Type;
   Cust_No :      Cust_No_Type;    -- Customer # NOT NULL
   Birthday :     Birthday_Type;   -- Customer birthday
   Birthday_TZ :  APQ_Timezone;     -- Timezone of the birthday
   Birthday_Ind : Boolean;         -- True when Birthday is NULL
begin
   ...
   Prepare(Q,"INSERT INTO BIRTHDAY (CUST_NO,BIRTHDAY)");
   Append(Q,"VALUES (");
   Append(Q,Cust_No,",");
   Encode(Q,Birthday,Birthday_TZ,Birthday_Ind,")" & New_Line);
   ...
```

If the Birthday_Ind indicator is false (not null), then the resulting query would look something like this:

```
INSERT INTO BIRTHDAY (CUST_NO,BIRTHDAY)
VALUES (12345,'1984-09-25 22:47:06+03')
```

The "+03" after the time represents the time zone UTC+3 hours.

3.3 Query Execution

Once the SQL query has been constructed using all of the techniques described in sections 3.1 and 3.2, you are ready to send the query to the database engine to have it executed. This is done with the help of the Query_Type's primitive Execute. The Execute call requires the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	Connection	in	out	Connection_Type'Class	-	The connection object

The Execute primitive can raise exceptions. They are summarized in the following table:

Exception Name	Reason
Not_Connected	There is no connection to use
Abort_State	Transaction in "abort state"
SQL_Error	The submitted SQL query failed

The Abort_State exception is described on page 68 (section 3.4). This exception indicates that the current transaction has failed. All other types of errors raise the SQL_Error exception, unless the connection is bad.

The use of the Execute primitive is illustrated by extending the example from page 56:

```
declare
   type Cust_No_Type is new Integer range 1000..100_000;
   type Birthday_Type is new APQ_Timestamp;
   procedure Append is new Append_Integer(Cust_No_Type);
   procedure Encode is new Encode_Integer(Cust_No_Type,Boolean);
   procedure Encode is new
      Encode_Timezone(Birthday_Type,APQ_Timezone,Boolean);
   C :            Connection_Type; -- Database connection
   Q :            Query_Type;      -- SQL Query object
   Cust_No :      Cust_No_Type;    -- Customer # NOT NULL
   Birthday :     Birthday_Type;   -- Customer birthday
   Birthday_TZ :  APQ_Timezone;     -- Timezone of the birthday
   Birthday_Ind : Boolean;         -- True when Birthday is NULL
begin
   ...
   Prepare(Q,"INSERT INTO BIRTHDAY (CUST_NO,BIRTHDAY)");
   Append(Q,"VALUES (");
   Append(Q,Cust_No,",");
   Encode(Q,Birthday,Birthday_TZ,Birthday_Ind,")" & New_Line);
   Execute(Q,C);
```

The example shows the Execute primitive pairing the query object *Q* with the database connection object *C*.

3.3.1 Error Message Reporting

It is well and fine to know that your SQL query failed, but more information is usually necessary. The Error_Message primitive can be invoked on the Query_Type object. This function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
	returns			String		The error message text

The following example shows how this function might be used:

```
...
begin
   Execute(Q,C);
exception
   when SQL_Error =>
      Put(Standard_Error,"SQL Error: ");
      Put_Line(Standard_Error,Error_Message(Q));
      raise;
   when others =>
```

```
        raise;
    end;
```

3.3.2 Is_Duplicate_Key Function

Duplicate key errors often occur while performing SQL INSERT operations on a table. A duplicate key insertion error is a special case because the insert operation may not be considered a failure when this happens, for some applications. For this reason, the Is_Duplicate_Key predicate function is provided for use after a SQL_Error exception has been raised[8]. The calling signature is documented as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
returns				Boolean		True if the Error_Message text indicates a duplicate key

The following example reports an SQL error only if the error is not a duplicate key insert problem:

```
...
begin
   Execute(Q,C);   -- Execute a INSERT SQL statement
exception
   when SQL_Error =>
      if not Is_Duplicate_Key(Q) then
         -- Only report error, if not a duplicate insert
         Put(Standard_Error,"SQL Error: ");
         Put_Line(Standard_Error,Error_Message(Q));
         raise;
      end if;
end;
```

3.3.3 Command_Status Function

The Command_Status function provides a string of status information after a query has been executed. The results returned depends upon the type of execution that was last performed. The following table summarizes the types of return status strings available:

This is a PostgreSQL specific function, only. Avoid its use for portability.

After Event	Result	Comments
CREATE ...	"CREATE"	
BEGIN WORK	"BEGIN"	
COMMIT WORK	"COMMIT"	
ROLLBACK WORK	"ROLLBACK"	
SELECT ...	"SELECT"	
INSERT ...	"INSERT <OID> <#>	# is normally 1

[8] At present, this test is implemented by calling Error_Message and looking at the message text. Future versions of the APQ binding may use a more reliable indicator if the PostgreSQL libpq library provides such a status indication.

Notice that after an INSERT is performed, the returned status string includes the OID (row ID) of the new row, and the number of rows inserted (normally 1). If you need to extract the OID value, see the Command_Oid function in section 3.3.4.

The calling signature of Command_Status is:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
returns				String		The command status text

The following exceptions are possible:

Exception Name	Reason
No_Result	There is no result status (no execution)

3.3.4 Command_Oid Function

After an INSERT operation, it may be necessary to know the Oid (row ID) value for the newly created row. While this can be extracted from the Command_Status return string (see section 3.3.3), the Command_Oid function makes this easier. The call signature for Command_Oid is as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
returns				Row_ID_Type		The OID of the inserted row

The following exceptions are possible:

Exception Name	Reason
No_Result	There is no result status (no execution)

Be aware that the exception No_Result can be raised for two different reasons:

- There was no prior “execution” (thus no result)
- The prior execution was not an INSERT operation (hence no OID value)

The following example shows how the INSERTed row’s OID values is obtained:

```
declare
   C :      Connection_Type;
   Q :      Query_Type;
   Obj_Id : Row_ID_Type;
begin
   ...
   Prepare(Q,”INSERT INTO CUST_ORDER (CATALOG_NO,QUANTITY,...”);
```

```
...
Execute(Q,C);
Obj_Id := Command_Oid(Q);  -- Get row id that was inserted
```

Portability Note:

Note that some databases do not encourage the use of a row ID. While PostgreSQL will support an Object ID (OID), MySQL will not return a row ID at all. Yet it is well recognized, that the application often needs to be able to identify specific rows within a table. In these cases, a serial value is encouraged instead.

MySQL requires that you use a serial value. To do this, you declare the table with a key value, that uses the non-standard SQL attribute "AUTO_INCREMENT" to provide a serial value. Here is an example:

```
CREATE TABLE MY_TABLE (
   ITEM_ID      INTEGER AUTO_INCREMENT NOT NULL PRIMARY KEY,
   DESCRIPTION  VARCHAR(80)
);
```

If your table includes an "AUTO_INCREMENT" key field like the one above, MySQL will return its generated serial value using the function Command_OID.

Generic_Command_Oid Function

To allow strong typing to be used in place of the supplied *Row_ID_Type* type, the Generic_Command_Oid function can be instantiated for use in the application. The instantiated function otherwise behaves identical with the Command_Oid function described on page 59. The instantiation arguments for Generic_Command_Oid are as follows:

#	Argument	Type	Description
1	Oid_Type	is new Row_ID_Type	The Specialized Oid type to use

An instantiation example follows:

```
declare
   type My_Oid_Type is new Row_ID_Type;
   function Command_Oid is new Generic_Command_Oid(My_Oid_Type);
```

3.3.5 Error Status Reporting

The Result function primitive is documented here for completeness. Applications should avoid using this function, since the values that it returns are very database technology specific.

PostgreSQL Result Codes

The PostgreSQL result types are declared in the APQ.PostgreSQL package. The Result_Type values are highly PostgreSQL engine specific and are enumerated in the following table:

Name	Value	Description
Empty_Query	0	The query returned 0 rows of data
Command_OK	1	The non-query statement executed successfully
Tuples_OK	2	The SQL query returned at least 1 row of data
Copy_Out	3	
Copy_In	4	
Bad_Response	5	Bad response from database server
Nonfatal_Error	6	A non fatal error has occurred
Fata_Error	7	A fatal error has occurred

Note that the numeric values thenselves are subject to change if the PostgreSQL database server software designers choose to do so. Use the enumerated names instead.

The Result function primitive that returns these values has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Result_Type		The result status (see above)

Notes:

- The Execute primitive will throw an exception if the execution failed (the Nonfatal_Error case).
- For SELECT queries, the fact that no rows are returned will be identifiable upon the first FETCH operation (the Empty_Query case), or upon calling End_of_Query.
- When rows are returned (the Tuples_OK case), the application will successfully fetch at least one row.
- For other SQL commands, successful execution is determined by Execute not throwing an exception (the Command_OK case).

For these reasons, applications should not normally require the use of the Result function.

MySQL Result Codes

The enumerated Result_Type type is declared in the package APQ.MySQL. The names of the different result codes are listed without their values below. This list may expand or shrink as the MySQL database development continues.

Code	Code
CR_NO_ERROR	ER_STACK_OVERRUN
ER_HASHCHK	ER_WRONG_OUTER_JOIN
ER_NISAMCHK	ER_NULL_COLUMN_IN_INDEX
ER_NO	ER_CANT_FIND_UDF
ER_YES	ER_CANT_INITIALIZE_UDF
ER_CANT_CREATE_FILE	ER_UDF_NO_PATHS
ER_CANT_CREATE_TABLE	ER_UDF_EXISTS
ER_CANT_CREATE_DB	ER_CANT_OPEN_LIBRARY
ER_DB_CREATE_EXISTS	ER_CANT_FIND_DL_ENTRY
ER_DB_DROP_EXISTS	ER_FUNCTION_NOT_DEFINED
ER_DB_DROP_DELETE	ER_HOST_IS_BLOCKED
ER_DB_DROP_RMDIR	ER_HOST_NOT_PRIVILEGED
ER_CANT_DELETE_FILE	ER_PASSWORD_ANONYMOUS_USER
ER_CANT_FIND_SYSTEM_REC	ER_PASSWORD_NOT_ALLOWED
ER_CANT_GET_STAT	ER_PASSWORD_NO_MATCH
ER_CANT_GET_WD	ER_UPDATE_INFO
ER_CANT_LOCK	ER_CANT_CREATE_THREAD
ER_CANT_OPEN_FILE	ER_WRONG_VALUE_COUNT_ON_ROW
ER_FILE_NOT_FOUND	ER_CANT_REOPEN_TABLE
ER_CANT_READ_DIR	ER_INVALID_USE_OF_NULL
ER_CANT_SET_WD	ER_REGEXP_ERROR
ER_CHECKREAD	ER_MIX_OF_GROUP_FUNC_AND_FIELDS
ER_DISK_FULL	ER_NONEXISTING_GRANT
ER_DUP_KEY	ER_TABLEACCESS_DENIED_ERROR
ER_ERROR_ON_CLOSE	ER_COLUMNACCESS_DENIED_ERROR
ER_ERROR_ON_READ	ER_ILLEGAL_GRANT_FOR_TABLE
ER_ERROR_ON_RENAME	ER_GRANT_WRONG_HOST_OR_USER
ER_ERROR_ON_WRITE	ER_NO_SUCH_TABLE
ER_FILE_USED	ER_NONEXISTING_TABLE_GRANT
ER_FILSORT_ABORT	ER_NOT_ALLOWED_COMMAND
ER_FORM_NOT_FOUND	ER_SYNTAX_ERROR
ER_GET_ERRNO	ER_DELAYED_CANT_CHANGE_LOCK
ER_ILLEGAL_HA	ER_TOO_MANY_DELAYED_THREADS
ER_KEY_NOT_FOUND	ER_ABORTING_CONNECTION
ER_NOT_FORM_FILE	ER_NET_PACKET_TOO_LARGE
ER_NOT_KEYFILE	ER_NET_READ_ERROR_FROM_PIPE
ER_OLD_KEYFILE	ER_NET_FCNTL_ERROR
ER_OPEN_AS_READONLY	ER_NET_PACKETS_OUT_OF_ORDER
ER_OUTOFMEMORY	ER_NET_UNCOMPRESS_ERROR
ER_OUT_OF_SORTMEMORY	ER_NET_READ_ERROR
ER_UNEXPECTED_EOF	ER_NET_READ_INTERRUPTED
ER_CON_COUNT_ERROR	ER_NET_ERROR_ON_WRITE

ER_OUT_OF_RESOURCES	ER_NET_WRITE_INTERRUPTED
ER_BAD_HOST_ERROR	ER_TOO_LONG_STRING
ER_HANDSHAKE_ERROR	ER_TABLE_CANT_HANDLE_BLOB
ER_DBACCESS_DENIED_ERROR	ER_TABLE_CANT_HANDLE_AUTO_INCREMENT
ER_ACCESS_DENIED_ERROR	ER_DELAYED_INSERT_TABLE_LOCKED
ER_NO_DB_ERROR	ER_WRONG_COLUMN_NAME
ER_UNKNOWN_COM_ERROR	ER_WRONG_KEY_COLUMN
ER_BAD_NULL_ERROR	ER_WRONG_MRG_TABLE
ER_BAD_DB_ERROR	ER_DUP_UNIQUE
ER_TABLE_EXISTS_ERROR	ER_BLOB_KEY_WITHOUT_LENGTH
ER_BAD_TABLE_ERROR	ER_PRIMARY_CANT_HAVE_NULL
ER_NON_UNIQ_ERROR	ER_TOO_MANY_ROWS
ER_SERVER_SHUTDOWN	ER_REQUIRES_PRIMARY_KEY
ER_BAD_FIELD_ERROR	ER_NO_RAID_COMPILED
ER_WRONG_FIELD_WITH_GROUP	ER_UPDATE_WITHOUT_KEY_IN_SAFE_MODE
ER_WRONG_GROUP_FIELD	ER_KEY_DOES_NOT_EXITS
ER_WRONG_SUM_SELECT	ER_CHECK_NO_SUCH_TABLE
ER_WRONG_VALUE_COUNT	ER_CHECK_NOT_IMPLEMENTED
ER_TOO_LONG_IDENT	ER_CANT_DO_THIS_DURING_AN_TRANSACTION
ER_DUP_FIELDNAME	ER_ERROR_DURING_COMMIT
ER_DUP_KEYNAME	ER_ERROR_DURING_ROLLBACK
ER_DUP_ENTRY	ER_ERROR_DURING_FLUSH_LOGS
ER_WRONG_FIELD_SPEC	ER_ERROR_DURING_CHECKPOINT
ER_PARSE_ERROR	ER_NEW_ABORTING_CONNECTION
ER_EMPTY_QUERY	ER_DUMP_NOT_IMPLEMENTED
ER_NONUNIQ_TABLE	ER_FLUSH_MASTER_BINLOG_CLOSED
ER_INVALID_DEFAULT	ER_INDEX_REBUILD
ER_MULTIPLE_PRI_KEY	ER_MASTER
ER_TOO_MANY_KEYS	ER_MASTER_NET_READ
ER_TOO_MANY_KEY_PARTS	ER_MASTER_NET_WRITE
ER_TOO_LONG_KEY	ER_FT_MATCHING_KEY_NOT_FOUND
ER_KEY_COLUMN_DOES_NOT_EXITS	ER_LOCK_OR_ACTIVE_TRANSACTION
ER_BLOB_USED_AS_KEY	ER_UNKNOWN_SYSTEM_VARIABLE
ER_TOO_BIG_FIELDLENGTH	ER_CRASHED_ON_USAGE
ER_WRONG_AUTO_KEY	ER_CRASHED_ON_REPAIR
ER_READY	ER_WARNING_NOT_COMPLETE_ROLLBACK
ER_NORMAL_SHUTDOWN	ER_TRANS_CACHE_FULL
ER_GOT_SIGNAL	ER_SLAVE_MUST_STOP
ER_SHUTDOWN_COMPLETE	ER_SLAVE_NOT_RUNNING
ER_FORCING_CLOSE	ER_BAD_SLAVE
ER_IPSOCK_ERROR	ER_MASTER_INFO
ER_NO_SUCH_INDEX	ER_SLAVE_THREAD
ER_WRONG_FIELD_TERMINATORS	ER_TOO_MANY_USER_CONNECTIONS

ER_BLOBS_AND_NO_TERMINATED	ER_SET_CONSTANTS_ONLY
ER_TEXTFILE_NOT_READABLE	ER_LOCK_WAIT_TIMEOUT
ER_FILE_EXISTS_ERROR	ER_LOCK_TABLE_FULL
ER_LOAD_INFO	ER_READ_ONLY_TRANSACTION
ER_ALTER_INFO	ER_DROP_DB_WITH_READ_LOCK
ER_WRONG_SUB_KEY	ER_CREATE_DB_WITH_READ_LOCK
ER_CANT_REMOVE_ALL_FIELDS	ER_WRONG_ARGUMENTS
ER_CANT_DROP_FIELD_OR_KEY	ER_NO_PERMISSION_TO_CREATE_USER
ER_INSERT_INFO	ER_UNION_TABLES_IN_DIFFERENT_DIR
ER_INSERT_TABLE_USED	ER_LOCK_DEADLOCK
ER_NO_SUCH_THREAD	ER_TABLE_CANT_HANDLE_FULLTEXT
ER_KILL_DENIED_ERROR	ER_CANNOT_ADD_FOREIGN
ER_NO_TABLES_USED	ER_NO_REFERENCED_ROW
ER_TOO_BIG_SET	ER_ROW_IS_REFERENCED
ER_NO_UNIQUE_LOGFILE	CR_UNKNOWN_ERROR
ER_TABLE_NOT_LOCKED_FOR_WRITE	CR_SOCKET_CREATE_ERROR
ER_TABLE_NOT_LOCKED	CR_CONNECTION_ERROR
ER_BLOB_CANT_HAVE_DEFAULT	CR_CONN_HOST_ERROR
ER_WRONG_DB_NAME	CR_IPSOCK_ERROR
ER_WRONG_TABLE_NAME	CR_UNKNOWN_HOST
ER_TOO_BIG_SELECT	CR_SERVER_GONE_ERROR
ER_UNKNOWN_ERROR	CR_VERSION_ERROR
ER_UNKNOWN_PROCEDURE	CR_OUT_OF_MEMORY
ER_WRONG_PARAMCOUNT_TO_PROCEDURE	CR_WRONG_HOST_INFO
ER_WRONG_PARAMETERS_TO_PROCEDURE	CR_LOCALHOST_CONNECTION
ER_UNKNOWN_TABLE	CR_TCP_CONNECTION
ER_FIELD_SPECIFIED_TWICE	CR_SERVER_HANDSHAKE_ERR
ER_INVALID_GROUP_FUNC_USE	CR_SERVER_LOST
ER_UNSUPPORTED_EXTENSION	CR_COMMANDS_OUT_OF_SYNC
ER_TABLE_MUST_HAVE_COLUMNS	CR_NAMEDPIPE_CONNECTION
ER_RECORD_FILE_FULL	CR_NAMEDPIPEWAIT_ERROR
ER_UNKNOWN_CHARACTER_SET	CR_NAMEDPIPEOPEN_ERROR
ER_TOO_MANY_TABLES	CR_NAMEDPIPESETSTATE_ERROR
ER_TOO_MANY_FIELDS	CR_CANT_READ_CHARSET
ER_TOO_BIG_ROWSIZE	CR_NET_PACKET_TOO_LARGE

The application should avoid direct reference to these database specific codes, where possible.

3.3.6 Generic APQ.Result

To enable generic database processing, APQ version 2.0 adds a new API function which is declared at the APQ.Root_Query_Type object level. This function returns a Natural result:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Natural		The result status (see above)

The value returned, represents the Result_Type'Pos(arg). In generic database code, you could use this generic function to retrieve the value. Later it can be turned into the appropriate Result_Type if required by doing a conversion. The following example illustrates:

```
with APQ.MySQL.Client, APQ.PostgreSQL.Client;
...
procedure App(Q : Root_Query_Type'Class) is
   R :    Natural;
   PQ_R : APQ.PostgreSQL.Result_Type;
   My_R : APQ.MySQL.Result_Type;
begin
   ...
   R := APQ.Result(Q);
   if APQ.Engine_Of(Q) = Engine_MySQL then
      My_R := APQ.MySQL.Result_Type'Val(R);
   ...
```

The code above demonstrates how generic database code is able to test for specific database error codes, when it is necessary.

3.3.7 Generic APQ.Engine_Of

As seen in the example of section 3.3.6, it is sometimes necessary to determine in generic code, what database technology is being used. Once this fact is known, the correct more specific action can be taken. Or in some cases, special actions must be performed in addition to the norm, for certain database technologies.

The function primitive **Engine_Of** can be used to determine the database technology being used:

#	Argument	in	out	Type	Default	Description
1	Q	in		Root_Query_Type	-	The SQL query object
returns				Database_Type		The database engine used

3.3.8 Checked Execution

For many utility programs where error reporting and recovery have simple requirements, a more compact and convenient way to execute queries can be applied. With checked execution, the query is not only executed, but any SQL errors are intercepted and reported to Standard_Error automatically. This saves the programmer effort when writing simple utility programs. Once the SQL_Error exception is intercepted and reported, the exception is re-thrown to leave control in the caller's realm. The important

thing here is that the error is caught and reported.

The Execute_Checked primitive has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	Connection	in	out	Connection_Type'Class	-	The connection object
3	Msg	in		String	""	Error message text

When the argument Msg is a non null length string like "Dropping table temp_tbl", the error message reported will be of the following format:

```
*** SQL ERROR: Dropping table temp_tbl
[FATAL_ERROR: ERROR: Relation "temp_tbl" does not exist]
```

The first line just identifies the fact that an SQL error occurred, and reports the *Msg* text. The second line first reports Result_Type'Image of the error, and then reports the error message text as returned by Error_Message. In this case, the example shows that Result_Type Fatal_Error was returned, and the error message returned from the database server was "ERROR: Relation "temp_tbl" does not exist".

When the null string (or the default value for the parameter) is given to argument Msg, then the SQL query is dumped out to Standard_Error instead. This is often useful for debugging purposes.

Changing the example found on page 56 slightly, we can apply the Execute_Checked primitive in the place of the Execute call.

```
declare
   type Cust_No_Type is new Integer range 1000..100_000;
   type Birthday_Type is new APQ_Timestamp;
   procedure Append is new Append_Integer(Cust_No_Type);
   procedure Encode is new Encode_Integer(Cust_No_Type,Boolean);
   procedure Encode is new
      Encode_Timezone(Birthday_Type,APQ_Timezone,Boolean);
   C :            Connection_Type; -- Database connection
   Q :            Query_Type;      -- SQL Query object
   Cust_No :      Cust_No_Type;    -- Customer # NOT NULL
   Birthday :     Birthday_Type;   -- Customer birthday
   Birthday_TZ :  APQ_Timezone;     -- Timezone of the birthday
   Birthday_Ind : Boolean;         -- True when Birthday is NULL
begin
   ...
   Prepare(Q,"INSERT INTO BIRTHDAY (CUST_NO,BIRTHDAY)");
   Append(Q,"VALUES (");
   Append(Q,Cust_No,",");
   Encode(Q,Birthday,Birthday_TZ,Birthday_Ind,")" & New_Line);
   Execute_Checked(Q,C);  -- Report errors if SQL_Error is raised
```

3.3.9 Suppressing Checked Exceptions

For utility work, it is sometimes convenient to have Execute_Checked report errors, but not raise SQL_Error. This is useful when you don't care about the outcome but want the error to be reported when detected. The raising or not raising of SQL_Error can be

controlled for the Execute_Checked primitive by calling Raise_Exceptions. It has the following calling requirements:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	Raise_On	in		Boolean	True	Enable/Disable raising SQL_Error

The following example shows how exceptions can be suppressed:

```
declare
   C :            Connection_Type; -- Database connection
   Q :            Query_Type;      -- SQL Query object
begin
   ...
   Raise_Exceptions(Q,False);      -- Suppress SQL_Error exception
   Execute_Checked(Q,C);           -- Report errors only
   Raise_Exceptions(Q,True);       -- Re-enable SQL_Error exceptions
```

3.3.10 Suppressing Checked Reports

Occaisionally, it is useful to control whether or not reporting is performed in the event of an SQL_Error. The reporting of errors can be controlled by the Report_Errors primitive procedure:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	Report_On	in		Boolean	True	Enable/Disable reporting SQL_Error

The default behaviour of a Query_Type is to report errors and raise SQL_Error when Execute_Checked experiences an SQL_Error exception. The reporting behaviour can be disabled as follows:

```
declare
   C :            Connection_Type; -- Database connection
   Q :            Query_Type;      -- SQL Query object
begin
   ...
   Report_Errors(Q,False);         -- Suppress error reporting
   Execute_Checked(Q,C);
```

Normally application programmers would not use Execute_Checked with error reporting disabled. However, it may be useful as a temporary measure to cause error reporting while debugging a program. Once the debugging has been completed, a global Boolean value could be set to false to prevent these errors from being reported.

3.4 Transaction Operations

Transaction operations consist of:

- BEGIN WORK
- COMMIT WORK
- ROLLBACK WORK

It is possible to build your own queries to accomplish these operations but the programmer is encourage to use the primitive operations below instead. One reason for using the APQ provided functions is to make your application portable to different databases. Sometimes there are slight variations on the SQL syntax required for the purpose. Additionally, it may be possible in the future to query the state of the transaction.[9]

The three primitives are named according to function :

Primitive Name	SQL Function
Begin_Work	BEGIN WORK
Commit_Work	COMMIT WORK
Rollback_Work	ROLLBACK WORK

Each of these primitives have the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	Connection	in	out	Connection_Type'Class	-	The connection object

These primitives will raise the following exceptions:

Exception Name	Reason
Not_Connected	There is no connection
Abort_State	A ROLLBACK is required
SQL_Error	This should not normally occur

The Abort_State[10] exception indicates that the database was in a transaction[11] when a processing error occurred (like a duplicate key on insert error). Once an error is encountered within a transaction, the only course to recovery is by executing a ROLLBACK WORK statement (this is done by the Rollback_Work call shown above). If duplicate inserts may occur, you must test for them in advance of the INSERT, to avoid placing the transaction into the "abort state".

The "abort state" itself is maintained in the Connection_Type object, causing the state to influence all Query_Type objects using the same connection. To clear the status, you must perform a Rollback_Work call on any Query_Type object, using the affected Connection_Type object where the status is saved.

The Query_Type object is used to form the SQL statement and to hold the result status. In application programming, you may want to dedicate one Query_Type object

[9] It is likely that a function like an In_Transaction function will be added at a future date.

[10] The abort state is currently unique to PostgreSQL. MySQL tolerates failed steps within a transaction.

[11] A "BEGIN WORK" statement was executed.

for each transaction in progress.[12] The following simple example demonstrates the use of these primitives:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   ...
   Begin_Work(Q,C);
   ...
   Commit_Work(Q,C);
```

3.5 Fetch Operations

Some database operations, particularly SELECT, return results. There are two fetch related primitives:

1. Sequential row fetch
2. Random access row fetch

The sequential fetch permits serial access of the resulting rows (tuples). Random access fetching permits rapid access to particular row results.

3.5.1 Fetch Limitations

Some databases like PostgreSQL have *no* limitations on how row data is fetched. The fetch may be sequential or random, as the application requires. Some other databases however, require some planning by the application programmer in this area. This distinction, and the API to control this problem is new to APQ 2.0, for database engines that require it.

For example, MySQL retrieves row data into the client program's address space in one of two ways:

- one row at a time, but all rows must be fetched
- all rows are loaded into client memory, for random access by the application

For large result sets, fetching one row[13] at a time is very practical. However, MySQL requires that the program fetch *all* row data. If the result set is large, and only an initial number of rows are required, this can be a serious performance issue.

When random access[14] of rows is required, MySQL requires that all row data be retrieved and stored into the client program. Fetching all of this data into client memory can be impractical for size reasons when the number of rows are large (there is an SQL work-around for this).

[12] This will be important later, if you want to query whether or not you are in a transaction.
[13] This is done using the mysql_use_result() function.
[14] This is done using the mysql_store_result() function.

The default APQ query mode is to assume that random access will be required. Note that sequential access is always permitted, even in random access mode (APQ hides this complication). If the application programmer is using a database that is limited in this way (MySQL), and has determined that fetching all results into client memory is not suitable, then the mode of the Query_Type needs to be changed by the program to use sequential access instead. See the next few sections on how to control the fetch query mode.

If you are only planning to use PostgreSQL, you can effectively ignore the sections about Fetch Query modes. However, if you plan to write your application in a database generic sort of way, you need to plan for the fetch query modes in your code.

3.5.2 Fetch Query Modes

Due to the performance limitations of different database engines (see preceding section), APQ provides the application programmer a way to control the fetch query mode. Package APQ defines the following type for this purpose:

```
type Fetch_Mode_Type is (
   Sequential_Fetch,
   Random_Fetch
);
```

By default, APQ assumes that the application programmer will need random access to row data. Hence the mode in effect is Random_Fetch by default. The **Fetch_Mode** function returns the current state of the Query_Type object:

#	Argument	in	out	Type	Default	Description
1	Q	in		Root_Query_Type	-	The SQL query object
returns				Fetch_Mode_Type		The fetch mode in effect

To change the current mode in effect, use the function **Set_Fetch_Mode** :

#	Argument	in	out	Type	Default	Description
1	Q	in	out	Root_Query_Type	-	The SQL query object
2	Mode	in		Fetch_Mode_Type	-	The new fetch mode

The application should only change the query mode *prior* to the *execution* of the query. When **Execute** or **Execute_Checked** are called, APQ must commit to the method that rows are being fetched. For this reason, set the query mode when the Query_Type is initially declared, after a call to **Reset** or **Prepare**, or prior to calling **Execute.**

The following exceptions may be raised by **Set_Fetch_Mode** :

Exception Name	Reason
Failed	Query results exist - cannot change mode

3.5.3 Sequential Fetch

The *Query_Type* object is always positioned at the first row after the query has been executed. Sequential fetches can then be performed to retrieve the first through to the last resulting row. The sequential **Fetch** primitive has the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object

A sequential fetch can always be made, whether the query object is in sequential or random mode. However, be aware that some databases (MySQL) require that all rows be fetched when in **Sequential_Fetch** mode. Note that the APQ default is to fetch in **Random_Fetch** mode. See section 3.5.2 to change this.

The Fetch primitive can raise the following exceptions:

Exception Name	Reason
No_Result	There was no command executed
No_Tuple	There were no result rows returned

The *No_Result* exception is raised when the *Query_Type* object is in the wrong state. For example, if the *Query_Type* object is cleared, and/or an SQL query is built but not Executed, then a *No_Result* exception will be raised.

The *No_Tuple* exception is raised when a Fetch is attempted when no resulting rows were returned by the database server. This exception can be avoided by calling the information function *End_of_Query* first, to see if there are any more rows to fetch. The *End_of_Query* function will be documented later. The following example shows how a sequential fetch is used:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   while not End_of_Query(Q) loop
      Fetch(Q);
      ...
   end loop;
   Clear(Q);
```

Clearing the query (or allowing it to fall out of scope) is recommended. This releases resources that are holding any prior query results.

3.5.4 Random Fetch

APQ assumes that random row fetches will be required by the application by default. The random fetch operation requires the use of the *Tuple_Index_Type* defined in the

package APQ:

```
type Tuple_Index_Type is new Positive;
```

Some databases like MySQL, have special fetch requirements. For those databases, you must ensure that the Query_Type object is in **Random_Fetch** mode (default). See section 3.5.2 to see how to query and set the fetch mode for these types of database engines.

The random **Fetch** primitive has the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	The SQL query object
2	TX	in		Tuple_Index_Type	-	The row # to fetch

Possible exceptions include:

Exception Name	Reason
No_Result	There was no command executed
No_Tuple	There were no result rows returned

A random fetch example is provided below:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   for TX in 1..Tuple_Index_Type(Tuples(Q)) loop
      Fetch(Q,TX);
      ...
   end loop;
   Clear(Q);
```

The function Tuples(Q) that was used in the *for* loop, returns the number of result rows for the query.[15] A slight modification of this loop could permit processing the rows in reverse order.

Notes:

1. If the tuple index *TX* provided to Fetch is out of range for the result set, no exception will be raised. An exception *will* be raised however, when the application attempts to fetch any value from that out of range row.

2. Any subsequent sequential fetch operation will fetch the row following the last randomly accessed row.

[15] Which can be zero.

3.5.5 Function End_of_Query

This function is depreciated. Catch the No_Tuple exception instead for greater database portability.

To facilitate sequential fetch operations, the End_of_Query primitive function has been provided. You have already seen this function used with the sequential fetch example on page 71. The calling requirements are summarized in the following table:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Boolean		True when no more rows

The function can raise the following exceptions:

Exception Name	Reason
No_Result	There was no command executed

The End_of_Query function returns a Boolean result:

False there is at least one more result row available (not at end)

True there are no more rows available (at end)

MySQL Note

The MySQL implementation of **End_Of_Query** is not a good one and so End_Of_Query should be avoided entirely in new code. The problem is located in the MySQL C API that is provided. The C mysql_eof() function returns false after reading the last row. It is only by fetching one more row and discovering that there are no more rows, that mysql_eof() then starts to return true. In other words, it returns true, when the end has already been reached. Since there is no way to work around this problem in MySQL, a generic database developer should avoid using End_Of_Query completely.

Catch the exception No_Tuple instead, when fetching rows.

3.5.6 Function Tuple

The Tuple function primitive is an information function that returns the current tuple number that was last fetched. If there has been no fetch yet, the *No_Tuple* exception is raised. The calling signature is as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Tuple_Index_Type		The last row number fetched

The exceptions that are possible include:

Exception Name	Reason
No_Tuple	There was no fetch performed yet

The following example shows the function being used:

```
declare
   C : Connection_Type;
   Q : Query_Type;
   X : Tuple_Index_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   while not End_of_Query(Q) loop
      Fetch(Q);
      TX := Tuple(Q);  -- Get Row #
      ...
   end loop;
   Clear(Q);
```

3.5.7 Rewind Procedure

Sometimes it is desireable to reprocess results sequentially. This is easily accomplished with the Rewind primitive. This primitive merely alters the state of the Query_Type object such that the next fetch operation will start with the first row.

The calling requirements are listed in the following table:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object

The following exceptions can be raised:

Exception Name	Reason
SQL_Error	The Query_Type is not in Random_Fetch mode

The following example shows the Rewind procedure being used:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   while not End_of_Query(Q) loop
      Fetch(Q);
      ...
   end loop;
   -- REPROCESS THE QUERY RESULTS :
   Rewind(Q);
   while not End_of_Query(Q) loop
```

```
      Fetch(Q);
      ...
   end loop;
   Clear(Q);
```

3.5.8 Tuples Function

You have already seen this function used in the example on page 72. This information function returns the number of result rows that are available. It should only be called after the *Query_Type* has been executed however. Otherwise the No_Result exception will be raised.

The calling signature for this function is summarized in the following table:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Natural		The number of result rows

The exceptions that are possible are summarized in the next table:

Exception Name	Reason
No_Result	There was no execute performed yet

For an example of use, see page 72.

3.6 Column Information Functions

After a query has been executed, which returns a set of rows, it is sometimes necessary to obtain information about the columns. Many of the functions make use of the following data type:

```
type Column_Index_Type is new Positive;
```

There are four column information functions:

Function Name	Purpose
Columns	Return the # of columns in each row
Column_Name	Return the column name for an index value
Column_Index	Return the index for a column name
Column_Type	Return type information for the column
Is_Null	Test if a column is null

All of these functions will raise the exception *No_Result* if an execute has not been performed successfully on the Query_Type object.

3.6.1 Function Columns

The Columns primitive function returns the number of columns available in each row of the result set. The calling arguments are summarized as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
returns				Natural		The number of result columns

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed

The following example shows the function in use:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,”SELECT CUST_NO,CUST_NAME”);
   Append(Q,”FROM CUSTOMER”);
   Execute(Q,C);
   while not End_of_Query(Q) loop
      Fetch(Q);
      for CX in 1..Column_Index_Type(Columns(Q)) loop
         ...process each column...
      end loop
   end loop;
   Clear(Q);
```

3.6.2 Function Column_Name

The primitive Column_Name returns the name of the column for a particular column index value. The calling requirements are summarized in the following table:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index number
returns				String		The column name

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	Bad Column_Index_Type value

The following example shows the function in use:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);

   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;

      for CX in 1..Column_Index_Type(Columns(Q)) loop
         Put_Line("Column Name: " & Column_Name(Q,CX));
      end loop
   end loop;
   Clear(Q);
```

3.6.3 Function Column_Index

If you have a column name, but want to know the column index value, then the Column_Index primitive function can be used. It's calling requirements are as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	Name	in		String	-	The column name
returns				Column_Index_Type		The column index

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	Unknown column name

The following rather contrived example shows the Column_Index function used in the pragma assert statement:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   while not End_of_Query(Q) loop
      Fetch(Q);
      for CX in 1..Column_Index_Type(Columns(Q)) loop
         declare
```

```
            Col_Name : String := Column_Name(Q,CX);
         begin
            Put_Line("Column Name: " & Col_Name);
            pragma assert(CX = Column_Index(Col_Name));
         end;
      end loop;
   end loop;
   Clear(Q);
```

3.6.4 Function Column_Type

PostgreSQL Type Information

The Column_Type primitive is the beginning of type information for the column. This function returns the Row_ID_Type value that describes the type in the pg_type PostgreSQL table. See the PostgreSQL database documentation for more details.

The Column_Type calling signature is as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
returns				Row_ID_Type		The pg_type Oid value

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	Unknown column name

MySQL Type Information

The field types supported by MySQL are defined by APQ.MySQL.Field_Type. The programmer may use the Query_Type primitive **APQ.MySQL.Client.Column_Type** to determine the column's type:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
returns				Field_Type		The MySQL field type value

At the writing of this manual, there were the following field types defined for MySQL:

Field_Type	MySQL Datatype
FIELD_TYPE_DECIMAL	DECIMAL
FIELD_TYPE_TINY	TINYINT | BOOLEAN
FIELD_TYPE_SHORT	SMALLINT

FIELD_TYPE_LONG	INTEGER
FIELD_TYPE_FLOAT	FLOAT
FIELD_TYPE_DOUBLE	DOUBLE
FIELD_TYPE_NULL	BOOLEAN
FIELD_TYPE_TIMESTAMP	TIMESTAMP
FIELD_TYPE_LONGLONG	BIGINT
FIELD_TYPE_INT24	MEDIUMINT
FIELD_TYPE_DATE	DATE
FIELD_TYPE_TIME	TIME
FIELD_TYPE_DATETIME	DATETIME
FIELD_TYPE_YEAR	YEAR
FIELD_TYPE_NEWDATE	?
FIELD_TYPE_ENUM	ENUM
FIELD_TYPE_SET	SET
FIELD_TYPE_TINY_BLOB	TINYTEXT | TINYBLOB
FIELD_TYPE_MEDIUM_BLOB	MEDIUMTEXT | MEDIUMBLOB
FIELD_TYPE_LONG_BLOB	LONGTEXT | LONGBLOB
FIELD_TYPE_BLOB	TEXT | BLOB
FIELD_TYPE_VAR_STRING	VARCHAR(N)
FIELD_TYPE_STRING	CHAR(N)

APQ does not yet fully support all of MySQL data types.

3.6.5 Is_Null Function

If a column is capable of returning a NULL value, it becomes necessary to test for this. The Is_Null calling arguments are as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
returns				Boolean		True if column is null

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	Unknown column name

The following example shows how to test if the CUST_NAME column is null or not:

```
declare
   C : Connection_Type;
   Q : Query_Type;
```

```
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,BIRTH_DATE");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      ...
      If Is_Null(Q,2) then
         -- CUST_NAME value is null
      end if;
      ...
   end loop;
   Clear(Q);
```

3.6.6 Column_Is_Null Generic Function

If you need to test for null using strongly typed indicators, you may want to instantiate the Column_Is_Null generic function. The generic parameters are:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean indicator type

This function is capable of the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates its use:

```
declare
   type Cust_Name_Ind_Type is new Boolean;
   function Is_Null is new Column_Is_Null(Cust_Name_Ind_Type);
   C : Connection_Type;
   Q : Query_Type;
   Cust_Name_Ind : Cust_Name_Ind_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      Cust_Name_Ind := Is_Null(Q,2);
```

```
      if not Cust_Name_Ind then
         -- Get Customer Name (since its not null)
      end if;
   end loop;
   Clear(Q);
```

3.7 Value Fetching Functions

Once a fetch operation has been performed, the application needs to retrieve the values for each column from the row. The function primitive **Value** assumes that the column's value is *not* going to be NULL. If it should be null however, the exception Null_Value is raised. A better set of primitives should be used for columns that may return NULL. See section 3.8.

The following subsections will cover the function primitives for extracting the values for builtin types.

3.7.1 Function Value

The value for a OID, string, bit string or Unbounded_String can be extracted for a column using the Value function primitive. This function is normally only used for columns that cannot return a NULL value.[16] The calling requirements for these primitives are the same with only the return type varying:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
returns				Row_ID_Type		Row ID value
				String		String column value
				APQ_Bitstring		Bit string value
				Ada.Strings.Unbounded. Unbounded_String		String column value

The following exceptions are possible:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index
Null_Value	The column's value is NULL

The following example shows how all the column values are fetched:

```
declare
   C : Connection_Type;
   Q : Query_Type;
begin
```

[16] If the value is NULL, the exception Null_Value will be raised.

```
Prepare(Q,"SELECT CUST_NO,CUST_NAME");
Append(Q,"FROM CUSTOMER");
Execute(Q,C);
loop
   begin
      Fetch(Q);
   exception
      when No_Tuple =>
         exit;
   end;
   for CX in 1..Column_Index_Type(Columns(Q)) loop
      declare
          Col_Value : String := Value(Q,CX);
      begin
          Put("Column");
          Put(Column_Index_Type'Image(CX));
          Put(" = "');
          Put(Value(Q,CX));
          Put_Line(""');
      end;
   end loop;
end loop;
Clear(Q);
```

3.7.2 Null_Oid Function

Since different database engines have different approaches to row ID values[17], it is necessary to know how to represent a null row ID value in the current database context. Use the **Null_Oid** primitive to obtain that value:

#	Argument	in	out	Type	Default
1	Query	in		Query_Type	-
	returns			Row_ID_Type	

Using the Null_Oid function in generic database code allows you to eliminate the test for the database engine being used, when comparing row ID values for null. Even non generic database code is well advised to use this function.

3.7.3 Generic Value Functions

The Value functions documented in section 3.7.1 were suitable for the specific data types that they supported. However, Ada programmers often derive distinct new types to prevent accidental mixing of values in expressions. To accomodate all of these custom data types, you need to use generic functions for the purpose:

Function Name	Data Type	Notes
Boolean_Value	is new Boolean	Any Boolean type
Integer_Value	is range <>	Any signed integer type

[17] MySQL for example, does not return row ID values.

Modular_Value	is mod <>	Any modular type
Float_Value	is digits <>	Any floating point type
Fixed_Value	is delta <>	Fixed point types
Decimal_Value	is delta <> digits <>	Any decimal type
Date_Value	is new Ada.Calendar.Time	Any date
Time_Value	is new Ada.Calendar.Day_Duration	Any time
Timestamp_Value	is new APQ_Timestamp	Time stamps

All of these generic functions accept the same instantiation parameters:

Argument Name	Data Type	Notes
Val_Type	<>	Type must correspond to generic function name
Ind_Type	is new Boolean	Any Boolean indicator type

These functions are capable of the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index
Null_Value	The column's value is NULL

The following example illustrates the use of the Integer_Value and Date_Value generic functions:

```
declare
   type Cust_No_Type is new APQ_Integer;
   type Cust_Birthday_Type is new APQ_Date;
   function Value is new Integer_Value(Cust_No_Type);
   function Value is new Date_Value(Cust_Birthday_Type);
   C : Connection_Type;
   Q : Query_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,BIRTH_DATE");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      declare
         Cust_Name : String := Value(Q,2); -- Col 2
         Cust_No :   Cust_No_Type;
         Birthday :  Cust_Birthday_Type;
      begin
         Cust_No  := Value(Q,1);  -- CUST_NO is col 1
         Birthday := Value(Q,3);  -- BIRTH_DATE is col 3
         ...
```

```
      end;
   end loop;
   Clear(Q);
```

In the example shown, *Cust_Name* is returned by the builtin function Value for String types. The variables *Cust_No* and *Birthday* are assigned through the generic instantiations of the functions Integer_Value and Date_Value respectively.

3.7.4 Fixed Length String Value Procedure

Sometimes in an application it is desireable to work with fixed length string values. The following Value procedure does just this:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	String	-	The string receiving the value

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index
Null_Value	The column's value is null

The following example extracts the CUST_NAME column result into variable *Cust_Name* as a 30 byte string value:

```
declare
   C :         Connection_Type;
   Q :         Query_Type;
   Cust_Name : String(1..30);
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      ...
      Value(Q,2,Cust_Name);
      ...
   end loop;
   Clear(Q);
```

3.7.5 APQ_Timezone Value Procedure

A TIMESTAMP value that carries with it a time zone value, requires a special procedure since two values must be extracted for the same column:

- The Ada.Calendar.Time[18] value holding the TIMESTAMP value
- The APQ_Timezone value holding the time zone offset

The generic procedure requires the following type arguments:

Argument Name	Data Type	Notes
Date_Type	is new Ada.Calendar.Time	Any Ada.Calendar.Time derived type
Zone_Type	is new APQ_Timezone	Any APQ_Timezone derived type

The signature of the instantiated procedure is as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	TS		out	APQ_Timestamp	-	The extracted TIMESTAMP value
4	TZ		out	APQ_Timezone	-	The extracted time zone offset value

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index
Null_Value	The column's value is null

The following example shows how the procedure can be used:

```
declare
   procedure Value is new Timezone_Value(APQ_Timestamp,APQ_Timezone);
   C : Connection_Type;
   Q : Query_Type;
   Birth_Date : APQ_Timestamp;
   Birth_Zone : APQ_Timezone;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,BIRTH_DATE");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
```

[18] From which APQ_Timestamp is derived.

```
      end;
      ...
      Value(Q,3,Birth_Date,Birth_Zone); -- Column BIRTH_DATE
      ...
      end loop;
   end loop;
   Clear(Q);
```

In this example code fragment, the Value procedure call expects the column's value to be column 3 (argument CX). This works as long as column BIRTH_DATE can never be null. If a NULL value is encountered in this program, the exception Null_Value will be raised and not caught by the code in this example.[19]

3.7.6 Bounded_Value Function

Bounded strings require a separate instantiation of Ada.Strings.Bounded for each string length. For this reason, a function supporting bounded strings must be provided in generic form. The **Bounded_Value** generic function accepts the following generic arguments:

Argument Name	Data Type	Notes
P	is new Ada.Strings.Bounded. Generic_Bounded_Length(<>)	Any bounded string instantiation

The resulting instantiated function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
returns				P.Bounded_String	-	The bounded string value

The function may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index
Null_Value	The column's value is null

The following example illustrates the use of Bounded_Value:

```
with Ada.Strings.Bounded;
declare
   package B30 is new Ada.Strings.Bounded.Generic_Bounded_Length(30);
   function Value is new Bounded_Value(B30);
   C :         Connection_Type;
```

[19] Either an exception handler must be added or a different way of extracting the value must be used. Fetch_Timezone is recommended if NULL is possible.

```
   Q :         Query_Type;
   Cust_Name : B30;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,BIRTH_DATE");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
      end;
      ...
      Cust_Name := Value(Q,2);
      ...
      end loop;
   end loop;
   Clear(Q);
```

3.8 Value and Indicator Fetch Procedures

The Value functions presented in section 3.7 were useful when the returned value was always going to be present. However, their use becomes clumsy and less efficient if exception handlers must be used to handle the NULL value case. This section documents Fetch procedures to return both a value and a null indicator together. With this convenience comes the added responsibility of checking the null indicator values, that are returned.

3.8.1 Char and Unbounded Fetch

The **Char_Fetch** and **Unbounded_Fetch** generic procedures fetch both a string value and an indicator value. In the Char_Fetch case, the returned value is blank filled to the full size of the receiving String buffer. Each of these has the following instantiation parameters:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean derived null indicator type

The resulting instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	String Ada.Strings.Unbounded. Unbounded_String	-	String Value
4	Indicator		out	Ind_Type	-	Indicator Value

The instantiated procedure may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates two different column fetch applications:

```
with Ada.Strings.Unbounded;
declare
   type Cust_Name_Ind_Type is new Boolean;
   type Cust_City_Ind_Type is new Boolean;
   subtype Cust_Name_Type is String(1..30);
   subtype Cust_City_Type is Ada.Strings.Unbounded.Unbounded_String;
   procedure Value is new Char_Fetch(Cust_Name_Ind_Type);
   procedure Value is new Unbounded_Fetch(Cust_City_Ind_Type);
   C :             Connection_Type;
   Q :             Query_Type;
   Cust_Name :     Cust_Name_Type;
   Cust_Name_Ind : Cust_Name_Ind_Type;
   Cust_City :     Cust_City_Type;
   Cust_City_Ind : Cust_City_Ind_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,CITY");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
      end;
      ...
      Value(Q,2,Cust_Name,Cust_Name_Ind);
      Value(Q,3,Cust_City,Cust_City_Ind);
      ...
   end loop;
   Clear(Q);
```

3.8.2 Varchar_Fetch and Bitstring_Fetch Procedures

To return a varying length string requires the use of a function. However, to return two values, we must resort to a procedure call for the purpose. In order to return both a varying length string and a null indicator, an additional return value is returned that indicates the length of the string. To accomodate strongly typed indicators, these two procedures are provided in generic form. The instantiation parameters are:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Any Boolean derived null indicator type

The resulting instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	String APQ_Bitstring	-	Receiving string buffer
4	Last		out	Natural	-	Length of returned string
5	Indicator		out	Ind_Type	-	Indicator Value

The instantiated procedure may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates two different column fetch applications:

```
declare
   type Cust_Name_Ind_Type is new Boolean;
   type Cust_City_Ind_Type is new Boolean;
   procedure Value is new Varchar_Fetch(Cust_Name_Ind_Type);
   procedure Value is new Varchar_Fetch(Cust_City_Ind_Type);
   C :              Connection_Type;
   Q :              Query_Type;
   Cust_Name :      String(1..30); -- Cust_Name(1..Cust_Name_Last)
   Cust_Name_Last : Natural;
   Cust_Name_Ind :  Cust_Name_Ind_Type;
   Cust_City :      String(1..40); -- Cust_City(1..Cust_City_Last)
   Cust_City_Last : Natural;
   Cust_City_Ind :  Cust_City_Ind_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME,CITY");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      ...
      Value(Q,2,Cust_Name,Cust_Name_Last,Cust_Name_Ind);
      Value(Q,3,Cust_City,Cust_City_Last,Cust_City_Ind);
      ...
   end loop;
   Clear(Q);
```

After the first Value call in the example, the customer name would be represented by the expression:

```
Cust_Name(1..Cust_Name_Last)
```

provided that the value *Cust_Name_Ind* was false.

3.8.3 Bounded_Fetch Procedure

To fetch both a Bounded_String value and its associated null indicator, you instantiate and call the **Bounded_Fetch**. The instantiation parameters are as follows:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Null indicator type
P	Ada.Strings.Bounded.Generic_Bounded_Length(<>)	Instantiation

The resulting instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	P.Bounded_String	-	Receiving Value
4	Indicator		out	Ind_Type	-	Indicator Value

The instantiated procedure may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates a fetch instantiation and call:

```
with Ada.Strings.Bounded;
declare
   package B32 is new Ada.Strings.Bounded.Generic_Bounded_Length(32);
   type Cust_Name_Ind_Type is new Boolean;
   procedure Value is new Bounded_Fetch(Cust_Name_Ind_Type,B32);
   C :              Connection_Type;
   Q :              Query_Type;
   Cust_Name :      B32;
   Cust_Name_Ind :  Cust_Name_Ind_Type;
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      ...
      Value(Q,2,Cust_Name,Cust_Name_Ind);
      ...
   end loop;
   Clear(Q);
```

3.8.4 Discrete Type Fetch Procedures

Several of the discrete types can be grouped and documented in this section. The following table indicates the generic procedure names and their associated class of data type for which they are designed:

Function Name	Data Type	Notes
Boolean_Fetch	is new Boolean	Any Boolean type
Integer_Fetch	is range <>	Any signed integer type
Modular_Fetch	is mod <>	Any modular type
Float_Fetch	is digits <>	Any floating point type
Fixed_Fetch	is delta <>	Fixed point types
Decimal_Fetch	is delta <> digits <>	Any decimal type
Date_Fetch	is new Ada.Calendar.Time	Any date
Time_Fetch	is new Ada.Calendar.Day_Duration	Any time
Timestamp_Fetch	is new APQ_Timestamp	Time stamps

Each of these generic procedures require the following generic parameters:

Argument Name	Data Type	Notes
Ind_Type	is new Boolean	Null indicator type

The resulting instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	<>	-	Type according to type class
4	Indicator		out	Ind_Type	-	Indicator Value

The instantiated procedure may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates how to instantiate and call a Integer_Fetch procedure.

```
declare
   type Cust_No_Type is new Integer;
   type Cust_No_Ind_Type is new Boolean;
   procedure Value is new Integer_Fetch(Cust_No_Type,Cust_No_Ind_Type);
   C :              Connection_Type;
   Q :              Query_Type;
   Cust_No :        Cust_No_Type;
   Cust_No_Ind :    Cust_No_Ind_Type;
```

```
begin
   Prepare(Q,"SELECT CUST_NO,CUST_NAME");
   Append(Q,"FROM CUSTOMER");
   Execute(Q,C);
   loop
      begin
         Fetch(Q);
      exception
         when No_Tuple =>
            exit;
      end;
      ...
      Value(Q,1,Cust_No,Cust_No_Ind);
      ...
   end loop;
   Clear(Q);
```

3.8.5 Timezone_Fetch Procedure

The Timezone_Fetch generic procedure is unique because it returns an additional parameter: the timezone. The procedure's instantiation parameters are listed below:

Argument Name	Data Type	Notes
Date_Type	is new Ada.Calendar.Time	The returned timestamp type
Zone_Type	is new APQ_Timezone	The returned timezone type
Int_Type	is new Boolean	The returned NULL indicator type

The resulting instantiated procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
3	V		out	Date_Type	-	Returned timestamp info
4	Z		out	Zone_Type	-	Returned timezone info
5	Indicator		out	Ind_Type	-	Indicator Value

The instantiated procedure may raise the following exceptions:

Exception Name	Reason
No_Result	There was no execute performed
No_Column	No column at index

The following example illustrates how to instantiate and call a Timezone_Fetch procedure.

```
declare
   type Bday_Type is new Integer;
   type Bday_Zone_Type is new APQ_Timezone;
   type Bday_Ind_Type is new Boolean;
```

```
      procedure Value is new
         Timezone_Fetch(Bday_Type,Bday_Zone_Type,Bday_Ind_Type);
      C :              Connection_Type;
      Q :              Query_Type;
      Bday :           Bday_Type;
      Bday_Zone :      Bday_Zone_Type;
      Bday_Ind :       Bday_Ind_Type;
   begin
      Prepare(Q,"SELECT CUST_NO,CUST_NAME,BIRTH_DATE");
      Append(Q,"FROM CUSTOMER");
      Execute(Q,C);
      loop
         begin
            Fetch(Q);
         exception
            when No_Tuple =>
               exit;
         end;
         ...
         Value(Q,3,Bday,Bday_Zone,Bday_Ind);
         ...
      end loop;
      Clear(Q);
```

3.9 Information Functions

You have already seen two information functions Result and Error_Message in section 3.3.1 and 3.3.5. Another useful Query_Type primitive is the To_String function. It is described in the next subsection.

3.9.1 The To_String Function

The To_String primitive allows the caller to retrieve the collected SQL text from the Query_Type object. The function's calling signature is as follows:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
	returns			String		String form of the entire query

The To_String function returns the full text of the SQL query, including newline characters.[20] If there has not been any SQL text collected, the function returns an empty string.[21]

The following example shows how a programmer can dump out the SQL query, but only when it fails:

```
   declare
      C :              Connection_Type;
      Q :              Query_Type;
```

[20] One is included at the end of the string if it is missing.

[21] In this case, no newline is provided.

```
begin
   Prepare(Q,”SELECT CUST_NO,CUST_NAME,BIRTH_DATE”);
   Append(Q,”FROM CUSTOMER”);
   begin
      Execute(Q,C);
   exception
      when SQL_Error =>
         Put_Line(“The failed SQL Query was:“);
         Put_Line(To_String(Q));
         raise;
      when others =>
         raise;
   end;
```

Chapter 4

Blob Support

The MySQL database provides a very different type of blob support. One that the author feels is somewhat inferior to the PostgreSQL functionality. However, this is an area for further study as far as APQ is concerned. For APQ 2.0, MySQL does not support blobs.

The PostgreSQL database however, like many others, provides the application programmer with the ability to store large amounts of information in a "blob". In many ways this resembles a file, with the exception that the contents are stored in the database and is accessed by number (OID). The APQ binding provides full PostgreSQL blob support for the Ada programmer. In addition, the Ada stream concept is employed to provide reliable and convenient access to the blob.

Endian Note:

The application programmer must keep in mind that any binary data written to a blob, by means of the Ada stream, is not endian neutral. This becomes a concern when a client application accesses or writes to blobs stored on a database over the network, on another host.

4.1 Introduction

Blob functions are managed primarily through the Blob_Type access type.[1] Stream I/O to and from the blob is performed using an Ada streams access value.[2]

The blob support can generally be grouped into the following categories:

- Create, Open and Close operations
- Index setting and querying operations

[1] The object itself is of type Blob_Object. Blob_Type is an access to Blob_Object type.

[2] Internally named Root_Stream_Access type, which is the type "access all Ada.Streams.Root_Stream_Type'Class".

- Information operations for Size and OID
- Stream accessor function
- Blob destruction
- File and Blob operations

The following sections will document the blob support using these groupings.

4.2 Blob Memory Leak Prevention

It is extremely important that the programmer realize that the Blob_Type data type is an access type.[3] Additionally this access value is a pointer to a dynamically allocated tagged record (type Blob_Object). For this reason, the programmer must take great care to "close" the Blob_Type before discarding the Blob_Type value, when it goes out of scope. Failure to close a Blob_Type value, will result in a memory leak and cause subsequent database performance issues. Use the Blob_Type value as if it were an open file that needs closing.

The following example represents a "Blob_Type leak":

```
declare
   C :   aliased Connection_Type;
   B :   Blob_Type;
begin
   ...
   B := Blob_Create(C'Access);
   ...
end;
```

The example above is bad because a "blob leak" occurs when the "end" statement is reached.[4] The variable *C* finalizes itself OK because it is a controled object.[5] However, *B* is an access type, pointing to a Blob_Object record. When variable *B* falls out of scope, only the pointer value in *B* is lost. The object it pointed to has not been released!

The following example code is better:

```
declare
   C :   aliased Connection_Type;
   B :   Blob_Type;
begin
   ...
   B := Blob_Create(C'Access);
   ...
   Blob_Close(B);
end;
```

[3] This design choice was necessary to accomodate Ada stream oriented I/O.

[4] It should also be noted that after creating a blob in the database, the application must save the OID value for the blob somewhere. Otherwise, you will have a blob in the database that will never be accessed!

[5] Both Connection_Type and Query_Type objects are controlled records with finalization.

The call to Blob_Close insures that the memory associated with the opened blob is released, before the value *B* falls out of scope.[6] However, if there is a chance that an exception may be raised, you may still be vulnerable to leaks. The following example covers all of the bases:

```
declare
   C :   aliased Connection_Type;
   B :   Blob_Type;
begin
   ...
   B := Blob_Create(C'Access);
   ...
   Blob_Close(B);
exception
   when others =>
      if B /= null then
         Blob_Close(B);
      end if;
      ...recovery steps...
end;
```

While the recovery steps have been left to the reader's imagination in the example above, the exception is caught and the value for variable *B* is tested. Only if *B* is not null, should the Blob_Close procedure call made. Using these principles, you can prevent blob memory leaks.

4.3 Create, Open and Close of Blobs

The most basic operations possible in an Ada program using blobs are:

- Creating a new blob in the database (Blob_Create)
- Opening an existing blob in the database (Blob_Open)
- Flushing buffered writes to the database (Blob_Flush)
- Closing a blob (Blob_Close)

The following subsections will explain how to perform these operations in detail.

4.3.1 Blob_Create Procedure

Before the application can open an existing blob, there must be some way to create a blob. The Blob_Create function does just this with the following calling signature:

Note: Blob operations must be performed within the context of a transaction.

#	Argument	in	out	Type	Default
1	DB	access		Connection_Type	-
2	Buf_Size	in		Natural	Buf_Size_Default
	returns			Blob_Type	

[6] Note that Close does not destroy the blob in the database.

The returned value is a Blob_Type that is capable of being used to read and/or write a blob. The blob is positioned at index position 1 (the beginning). See section 4.6.2 for information about how to determine the created blob's OID.

Starting with APQ version 1.2, all blob I/O is buffered if the *Buf_Size* argument is supplied with a value greater than zero, or is not supplied such that the default value applies. The following table summarizes the *Buf_Size* argument behaviour:

Buf_Size Value	Description	Performance
0	Unbuffered blob I/O	Very poor
0 < Buf_Size < 1024	Buffered	Poor
1024 <= Buf_Size < Buf_Size_Default	Buffered blob I/O	Better
Buf_Size_Default	Buffered: 5120 bytes	Very good

The following exceptions are possible:

Exception Name	Reason
Blob_Error	There was no blob created

Possible reasons for a Blob_Error exception to be raised would include:

- bad database connection object[7]
- a database error occurred (no more blob space?)

The following example shows how a new blob can be created:

```
declare
   C :   aliased Connection_Type;
   B :   Blob_Type;
begin
   ...
   B := Blob_Create(C'Access);
   ...
   Blob_Close(B);
end;
```

Note:

The argument *DB* in the call to the Blob_Create, is an access to Connection_Type argument. You must guarantee that the Connection_Type object does not finalize before the created blob has been closed.

4.3.2 Blob_Open Function

To open an existing blob, you must know the OID of the blob in the database. This is normally a value that is stored in a database column somewhere. See section 4.6.2 for information on how to determine the OID of a created blob.

Note: Blob operations must be performed within the context of a transaction.

[7] Or the database connection object went out of scope.

Blobs can be opened for various types of access:

Read for readonly access to the blob contents

Write for writing to the blob

Read_Write for both reading and writing of the blob

The Blob_Open function has the following calling signature:

#	Argument	in	out	Type	Default
1	DB	access		Connection_Type	-
2	Oid	in		Row_ID_Type	-
3	Mode	in		Mode_Type	-
4	Buf_Size	in		Natural	Buf_Size_Default
	returns			Blob_Type	

The returned value is a Blob_Type that is accessable according to the *Mode* selected. The blob is positioned at index value 1 (the beginning).

Starting with APQ version 1.2, all blob I/O is buffered if the *Buf_Size* argument is supplied with a value greater than zero, or is not supplied such that the default value applies. The following table summarizes the *Buf_Size* argument behaviour:

Buf_Size Value	Description	Performance
0	Unbuffered blob I/O	Very poor
0 < Buf_Size < 1024	Buffered	Poor
1024 <= Buf_Size < Buf_Size_Default	Buffered blob I/O	Better
Buf_Size_Default	Buffered: 5120 bytes	Very good

The following exceptions are possible:

Exception Name	Reason
Blob_Error	There was no blob opened

Possible reasons for a Blob_Error exception to be raised would include:

- bad database connection object
- the Oid value supplied is not known by the database
- a database error occurred

The following example shows how blob 73763 can be opened for reading:

```
declare
   C :   aliased Connection_Type;
   B :   Blob_Type;
```

```
   OID : Row_ID_Type := 73763;
begin
   ...
   B := Blob_Open(C'Access,OID,Read);
   ...
   Blob_Close(B);
exception
   when others =>
      if B /= null then
         Blob_Close(B);
      end if;
      raise;
end;
```

Note:

The argument *DB* in the call to the Blob_Open, is an access to Connection_Type argument. You must guarantee that the Connection_Type object does not finalize before the opened blob has been closed.

Generic_Blob_Open Function

To allow the application programmer to use strong types in place of the supplied Row_ID_Type type, a generic procedure for opening blobs is also provided. The instantiated function behaves exactly as described for Blob_Open on page 98. The instantiation arguments for Generic_Blob_Open are:

#	Argument	Type	Description
1	Oid_Type	is new Row_ID_Type	The Specialized Oid type to use

The following example shows how to instantiate the function:

```
declare
   type My_Oid_Type is new Row_ID_Type;
   function Blob_Open is new Generic_Blob_Open(My_Oid_Type);
```

4.3.3 Blob_Flush Procedure

When you are using buffered blob I/O[8] and your application has performed one or more writes to the blob, you may need to be certain that all of the buffered data is physically written out to the database. For example, you may have a timing opportunity to perform this expensive operation while the user is waiting for something else to occur. Buffer flushes are automatically performed when the blob is closed or due to changes made by the Blob_Set_Index operation. To give the application programmer control over the timing of the physical write to the database, the Blob_Flush procedure can be used.

Blob_Flush calls are ignored when unbuffered blob I/O is being used. This makes it easy for the application to choose buffered or unbuffered operation without source code changes.

The Blob_Flush procedure has the following calling signature:

[8] Buffered blob I/O is the default for performance reasons.

#	Argument	in	out	Type	Default
1	Blob	in	out	Blob_Type	-

The following exceptions are possible:

Exception Name	Reason
Blob_Error	The blob is not open

4.3.4 Blob_Close Procedure

When the programmer no longer requires access to a open/created blob, the procedure Blob_Close should be called. Since an open blob depends upon a hidden access value that points back to the Connection_Type object, the programmer should call Blob_Close as soon as is practical. This reduces the possibility of error that will occur if the Connection_Type object is finalized too soon.

The Blob_Close procedure has the following calling signature:

#	Argument	in	out	Type	Default
1	Blob	in	out	Blob_Type	-

The following exceptions are possible:

Exception Name	Reason
Blob_Error	The blob is not open

Normally the Blob_Error exception will indicate an attempt to close a blob that is not open. However, it is possible that the database engine may experience a problem that will raise the same exception.

The procedure Blob_Close will also null out the the Blob_Type value that was passed in. This is done to eliminate any accidental access to a Blob_Object that no longer exists.

4.4 Index Setting Operations

Like a file, a blob's "position" can be changed and queried. The index operations require the use of two types defined for the purpose. They are:

```
type Blob_Count is new Ada.Streams.Stream_Element_Offset
     range 0..Ada.Streams.Stream_Element_Offset'Last;

subtype Blob_Offset is Blob_Count range 1..Blob_Count'Last;
```

The type *Blob_Count* is used where there is a count involved (which may require the value zero). The type *Blob_Offset* is used whenever a blob offset is used, since it starts at the value 1.

The next subsections describe facilities for performing blob indexing operations.

4.4.1 Blob_Set_Index Procedure

The Blob_Set_Index procedure is used when the caller needs to seek to a new position within the opened blob. See section 4.6.1 if you need to know the size of the blob.

The calling requirements for Blob_Set_Index are summarized in the following table:

#	Argument	in	out	Type	Default
1	Blob	in		Blob_Type	-
2	To	in		Blob_Offset	-

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open or seek failed

The following example shows how to seek to the end of the blob:

```
declare
   C :        aliased Connection_Type;
   B :        Blob_Type;
   B_Size :   Blob_Count;
   End_Blob : Blob_Index;
begin
   ...
   B_Size := Blob_Size(B);
   if B_Size > 0 then
      End_Blob := B_Size;
      Blob_Set_Index(B,End_Blob);
   ...
```

4.5 Blob_Index Function

Applications sometimes need to query where they are positioned in the blob. The Blob_Index function returns the current Blob_Offset position information. The calling requirements are as follows:

#	Argument	in	out	Type	Default
1	Blob	in		Blob_Type	-
	returns			Blob_Offset	

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open

The following example code determines where in the presently opened blob the blob position index is:

```
declare
   C :        aliased Connection_Type;
   B :        Blob_Type;
   Pos_Blob : Blob_Index;
begin
   ...
   Pos_Blob := Blob_Index(B);
```

4.6 Information Functions

The following subsections describe information gathering functions. They provide the programmer a way to obtain size and identification information.

4.6.1 Blob Size Function

To determine the present size of a blob, the Blob_Size function can be used. It's calling signature is as follows:

#	Argument	in	out	Type	Default
1	Blob	in		Blob_Type	-
	returns			Blob_Count	

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open

Notice that the return type Blob_Count does permit the value zero to be returned (blob is empty).

The following example code determines the size of the presently opened blob:

```
declare
   C :         aliased Connection_Type;
   B :         Blob_Type;
   Blob_Size : Blob_Count;
begin
   ...
   Blob_Size := Blob_Size(B);
```

4.6.2 Blob_OID Function

After a blob is created, it is very necessary to determine the OID for the blob. The Blob_Oid function may be called after Blob_Create or Blob_Open to obtain OID information. The calling requirements are as follows:

#	Argument	in	out	Type	Default
1	Blob	in		Blob_Type	-
	returns			Row_ID_Type	

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open

The following example code determines the OID value for the newly created blob:

```
declare
   C :         aliased Connection_Type;
   B :         Blob_Type;
   Blob_OID :  Row_ID_Type;
begin
   ...
   B        := Blob_Create(C'Access);
   Blob_OID := Blob_Oid(B);
```

Generic_Blob_Oid Function

To use a strongly typed version of the Blob_Oid function, the application programmer can instantiate from Generic_Blob_Oid. The instantiated function otherwise behaves exactly as the Blob_Oid function on page 104. The instantiation parameters for Generic_Blob_Oid are:

#	Argument	Type	Description
1	Oid_Type	is new Row_ID_Type	The Specialized Oid type to use

The following example shows how to instantiate the function:

```
declare
    type My_Oid_Type is new Row_ID_Type;
    function Blob_Oid is new Generic_Blob_Oid(My_Oid_Type);
```

4.6.3 End_Of_Blob Function

The End_Of_Blob function can be used by programs that sequentially read through a blob. The calling signature for this function is given below:

Note that this function results in poor performance if the buffer size is set to zero (unbuffered) in the opening Blob_Open/Blob_Create calls.

#	Argument	in	out	Type	Default

1	Blob	in		Blob_Type	-
	returns			Boolean	

The return value is True if the current position in the blob is at the end of the blob. Otherwise the value False is returned.

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open

The following example code reads a series of strings from the blob, using the End_Of_Blob function:

```
declare
   C : aliased Connection_Type;
   B : Blob_Type;
begin
   ...
   B := Blob_Open(...);
   declare
      S : Root_Stream_Access := Blob_Stream(B);
   begin
      while not End_Of_Blob(B) loop
         declare
            Line : String := String'Input(S);
         begin
            Put_Line(Line);
         end;
      end loop;
   end;
   Blob_Close(B);
```

4.7 Stream Access

In order for an Ada program to perform stream I/O on a blob, you must obtain a useable stream pointer. The APQ binding defines a type named Root_Stream_Access for this purpose. It is defined as follows:

```
type Root_Stream_Access is access all Ada.Streams.Root_Stream_Type'Class;
```

The function Blob_Stream returns this stream pointer to the caller. Use of this returned pointer makes it possible to use the native Ada stream I/O facilities. The function Blob_Stream is outlined in the following table:

#	Argument	in	out	Type	Default
1	Blob	in		Blob_Type	-
	returns			Root_Stream_Access	

The following exceptions are possible:

Exception Name	Reason
Blob_Error	Not open

The following example shows how a blob is created, a stream access value is obtained, and a APQ_Timestamp value is written to the new blob:

```
declare
   C :         aliased Connection_Type;
   B :         Blob_Type;
   Str :       Root_Stream_Access;
   Some_Date : APQ_Timestamp;
begin
   ...
   B   := Blob_Create(C'Access);
   declare
      Str : Root_Stream_Access := Blob_Stream(B);
   begin
      APQ_Timestamp'Write(Str,Some_Date);
      ...
   end;
   Blob_Close(B);
end;
```

Notice in this example, that the declaration and the existance of the stream pointer *Str* was restricted as much as possible. While not strictly necessary, there are good reasons for following this practice. See the special following note for the details.

Note:

The programmer does not have to worry about "closing" or freeing the returned stream pointer (*Str* in the example). It can be nulled or left to fall out of scope. Only the type *Blob_Type* must be "closed" by calling Blob_Close.

The programmer must however be careful to never use the stream pointer after the blob has been closed, or after the connection object has been closed or finalized. The stream pointer should be nulled when it has outlived its usefulness, or allowed to fall out of scope.

4.8 Blob Destruction

To release a blob, you must use the Blob_Unlink procedure call. The calling arguments are summarized in the following table:

#	Argument	in	out	Type	Default
1	DB	in		Connection_Type	-
2	Oid	in		Row_ID_Type	-

The following exceptions are possible:

Exception Name	Reason
Blob_Error	No such Oid

The following code releases the blob referenced in the example on page 100.

```
declare
   C :   aliased Connection_Type;
   Oid : Row_ID_Type := 73763;
begin
   ...
   Blob_Unlink(C,Oid);  -- Destroy blob 73763
```

Generic_Blob_Unlink Procedure

To use a strongly typed version of the Blob_Unlink procedure, the application programmer can instantiate from Generic_Blob_Unlink. The instantiated function otherwise behaves exactly as the Blob_Unlink function. The instantiation parameters for Generic_Blob_Unlink are:

#	Argument	Type	Description
1	Oid_Type	is new Row_ID_Type	The Specialized Oid type to use

The following example shows how to instantiate the function:

```
declare
   type My_Oid_Type is new Row_ID_Type;
   procedure Blob_Unlink is new Generic_Blob_Unlink(My_Oid_Type);
```

4.9 File and Blob Operations

Blobs are very similar to files. It should be no surprise then that sometimes a file is imported into a blob, or exported from a blob.

A file is imported into a blob with the Blob_Import call:

#	Argument	in	out	Type	Default
1	DB	in		Connection_Type	-
2	Pathname	in		String	-
3	Oid		out	Row_ID_Type	-

Blob_Import returns the Oid of the newly created blob, that now contains a copy of the file specified by the Pathname argument.

A blob's contents can be written out (exported) to a file with a call to Blob_Export:

#	Argument	in	out	Type	Default

1	DB	in		Connection_Type	-
2	Oid	in		Row_ID_Type	-
3	Pathname	in		String	-

After a successful return from Blob_Export, the file named by the Pathname argument, contains a copy of the specified blob.

If any error in these import/export operations occur, the following exception is raised:

Exception Name	Reason
Blob_Error	Import/export failed

Note that Blob_Import creates a new blob if necessary. Blob_Export creates a new file if necessary.

Generic_Blob_Import and Generic_Blob_Export Procedures

To use strongly typed versions of the Blob_Import and Blob_Export, the application programmer can instantiate from Generic_Blob_Import and Generic_Blob_Export respectively. The instantiated procedure otherwise behaves exactly as the Blob_Import or Blob_Export function. The instantiation parameters for Generic_Blob_Import or Generic_Blob_Export are:

#	Argument	Type	Description
1	Oid_Type	is new Row_ID_Type	The Specialized Oid type to use

The following example shows how to instantiate the function:

```
declare
   type My_Oid_Type is new Row_ID_Type;
   procedure Blob_Import is new Generic_Blob_Import(My_Oid_Type);
```

Chapter 5

Utility Functions

5.1 To_String Support

To ease the job for the application developer, a number of builtin To_String functions are provided to allow conversion from the data type to its string representation. The following To_String functions are available with the following builtin types (only one function requires a second argument):

Argument 1 (V)	Argument 2 (TZ)
APQ_Boolean	
APQ_Date	
APQ_Time	
APQ_Timestamp	
APQ_Timestamp	APQ_Timezone
APQ_Bitstring	
APQ_Timezone	

The following illustrates one example:

```
declare
   Ship_Date : APQ_Date;
begin
   Put("Shipped on: ");
   Put_Line(To_String(Ship_Date));
```

5.2 Generic To_String Support

Programs that make use of distinct types will require the use of generic functions to perform To_String conversions. The following generic functions are available for instantiation:

Name	Argument	Parameter Type

Boolean_String	Val_Type	is new Boolean
Integer_String	Val_Type	is new range <>
Modular_String	Val_Type	is new mod <>
Fixed_String	Val_Type	is new delta <>
Float_String	Val_Type	is new digits <>
Decimal_String	Val_Type	is new delta <> digits <>
Date_String	Val_Type	is new Ada.Calendar.Time
Time_String	Val_Type	is new Ada.Calendar.Day_Duration
Timestamp_String	Val_Type	is new Ada.Calendar.Time
Timezone_String	Val_Type	is new APQ_Timezone

The instantiated function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	V	in		Val_Type	-	The value to convert
	returns			String		String result

The following example illustrates their use:

```
declare
   type My_Date_Type is new APQ_Timestamp;
   function To_String is new Timestamp_String(My_Date_Type);
   Execution_Date : My_Date_Type;
begin
   ...
   Put("Program Execution Date: ");
   Put_Line(To_String(Execution_Date));
```

5.3 Conversion Generic Functions

Sometimes a programmer must convert from a text format string into another data type for manipulation. Several generic functions are provided for the purpose:

Generic Name	Argument (S)	Parameter Type
Convert_To_Boolean	String	is new Boolean
Convert_To_Date	String	is new Ada.Calendar.Time
Convert_To_Time	String	is new Ada.Calendar.Day_Duration
Convert_To_Timestamp	String	is new Ada.Calendar.Time

These generic functions take one generic parameter Val_Type, representing the return type. The Val_Type must derive as the table above indicates. The instantiated function takes the following form:

#	Argument	in	out	Type	Default	Description
1	S	in		String	-	The value to convert

	returns	Val_Type		The conversion result

The following exceptions are possible:

Exception Name	Reason
Invalid_Format	The input value was not a proper value for the type

The following example illustrates some converions:

```
declare
   type Bool is new APQ_Boolean;
   type Birth_Date_Type is new APQ_Date;
   function To_Boolean is new Convert_To_Boolean(Bool);
   function To_Date is new Convert_To_Date(Birth_Date_Type);
   My_Bool : Bool;
   Elvis :   Birth_Date_Type;
begin
   ...
   My_Bool := To_Boolean(“F”);
   Elvis   := To_Timestamp(“1957-01-08”);
```

5.4 The Convert_Date_and_Time Generic Function

Sometimes the programmer needs the convenience of putting separate date and time values together into a returned timestamp value. For example the date of birth may be stored in one database column, while the time of birth is stored in another. It may be necessary to work with a timestamp value instead, that contains both of these components. To permit the use of strongly typed values, a generic function is provided for this purpose.

The generic inputs to Convert_Date_and_Time are:

Argument Name	Data Type	Notes
Date_Type	is new Ada.Calendar.Time	Type of input date value
Time_Type	is new Ada.Calendar.Day_Duration	Type of input time value
Result_Type	is new Ada.Calendar.Time	The type of the returned timestamp

The instantiated function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	DT	in		Date_Type	-	The input date value
2	TM	in		Time_Type	-	The input time value
	returns			Result_Type		The combined date and time

The following example shows how to apply this function:

```
declare
```

```
   type My_Date_Type is new APQ_Date;
   type My_Time_Type is new APQ_Time;
   type My_Timestamp_Type is new APQ_Timestamp;
   function To_Timestamp is new
      Convert_Date_and_Time(
         Date_Type => My_Date_Type,
         Time_Time => My_Time_Type,
         Result_Type => My_Timestamp_Type);
   Some_Date :      My_Date_Type;
   Some_Time :      My_Time_Type;
   Some_Timestamp : My_Timestamp_Type;
begin
   ...
   Some_Timestamp := To_Timestamp(Some_Date,Some_Time);
```

5.5 The Extract_Timezone Generic Procedure

When a database table or result column provides a timestamp and timezone together, it is sometimes necessary to extract these components so that they can be manipulated separately. To permit the use of application defined types, a generic procedure is provided. The Extract_Timezone generic procedure requires the following inputs:

Argument Name	Data Type	Notes
Date_Type	is new Ada.Calendar.Time	Type of output date value
Zone_Type	is new APQ_Timezone	Type of output timezone value

The instantiated function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	S	in		String	-	The input timestamp and zone value
2	DT		out	Date_Type	-	The output timestamp value
3	TZ		out	Zone_Type	-	The output timezone value

The following exceptions are possible:

Exception Name	Reason
Invalid_Format	The input value was not a proper value for the type

The following example shows how to apply this procedure:

```
declare
   type My_Date_Type is new APQ_Timestamp;
   type My_Zone_type is new APQ_Timezone;
   procedure Extract is new Extract_Timezone(My_Date_Type,My_Zone_Type);
   Ex_Date :      My_Date_Type;  -- Extracted timestamp
   Ex_Zone :      My_Zone_Type;  -- Extracted timezone
begin
   ...
   Extract("1957-01-08 01:13:45+04",Ex_Date,Ex_Zone);
```

Chapter 6

Calendar Functions

There is frequently the need in applications to separate out the hour, minute and second from a time value. To make this easier, and to permit the continued use of strong typing, the following generic functions are available:[1]

Generic Name	Return Type	Description
Generic_Hour	Hour_Number	Extracts the hour from time
Generic_Minute	Minute_Number	Extracts minute from time
Generic_Second	Second_Number	Extracts second from time

Any of these generic functions require the following generic parameters:

Argument Name	Data Type	Notes
Time_Type	is new Ada.Calendar.Day_Duration	Type of input time value

The instantiated function has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	TM	in		Time_Type	-	The input time value
	returns			*Unit*_Type		*Unit* = Hour, Minute or Second

The following example shows how to apply this procedure:

```
declare
   type Evt_Time_Type is new Ada.Calendar.Day_Duration;
   function Hour is new Generic_Hour(Evt_Time_Type);
   function Minute is new Generic_Minute(Evt_Time_Type);
   Evt_Time : Evt_Time_Type;
   HH :       Hour_Number;
   MM :       Minute_Number;
begin
```

[1] It could be argued that these generic functions do not go the full generic distance, because the return value types are standard types only (types Hour_Number, Minute_Number and Second_Number).

```
...
HH := Hour(Evt_Time);    -- Extract Hour
MM := Minute(Evt_Time);  -- Extract Minute
```

Chapter 7

Decimal Support

In order to support accurate number calculations, particularly for financial work, the package APQ.PostgreSQL.Decimal is available for the programmer to use. This package is based upon the C source code extracted out of the PostgreSQL server.[1] The decimal support is *not* a floating point package, but does support approximately 1,000 digits worth of accuracy. Note that this is currently PostgreSQL specific code, and as such, it has not been reworked for general use in databases like MySQL.

7.1 Introduction

The PostgreSQL.Decimal package is a binding to the extracted server decimal code. This gives the Ada programmer access to the same numeric support as used by the database engine to sum columns etc. Because it is decimal based, you will not have to worry about representation issues for values like 0.3,[2] allowing for accurate sums and hash total calculations.

Special Note: The PostgreSQL.Decimal package is still under development, and is subject to change. One of the most serious limitations at present is the fact that assignment clobbers any prior concept of precision and scale for the variable being assigned to. To overcome this, it is possible that a future implementation of this package may provide task safe storage to preserve the variable's precision and scale. This can be done by saving the variable's precision and scale in task safe storage at finalization time. When the Adjust primitive is later called, the saved precision and scale can be restored and followed by a call to the Constrain() function. This will implicitly keep the variable within its configured precision and scale parameters.

[1] Portions copyright (c) 1996-2001, The PostgreSQL Global Development Group, and portions copyright (c) 1994, The Regents of the University of California.

[2] In binary floating point, the value 0.3 must be represented as 0.29999 repeat.

7.2 Decimal Exceptions

The PostgreSQL.Decimal binding can raise any of the following set of exceptions:

Exception Name	Reason
Decimal_NaN	The value is “Not a Number“ (or value is NULL)
Decimal_Format	Input does not properly represent a decimal number
Decimal_Overflow	The value over/under-flowed.
Undefined_Result	The computation does not have a defined result
Divide_By_Zero	An attempt to divide by zero occurred

7.3 “Not a Number” Operations

A new Decimal_Type value is initialized to NaN (Not a Number). This is a special status for the value, which can be assigned to other Decimal_Type values. When this status is detected in an expression where a computation is being performed, the exception Decimal_NaN is raised to indicate that no valid result can be determined.

7.4 The Decimal_Type Type

Decimal values are manipulated in a type, which is defined in terms of a tagged controlled record:

```
type Decimal_Type is new Ada.Finalization.Controlled with private;
```

These values are further defined by the following additional two attributes:

Precision specifies the precision of the decimal variable

Scale specifies the scale of the decimal variable

7.5 Is_NaN Function

To test if a value is “Not a Number”, the Is_NaN function can be used:

#	Argument	in	out	Type	Default	Description
1	DT	in		Decimal_Type	-	The input decimal value
	returns			Boolean		True if the value is “Not a Number“

The following example shows how to apply the function:

```
declare
   D : Decimal_Type;
begin
```

```
if Is_NaN(D) then
   Put_Line("D is NaN!");
   ...
```

7.6 Convert Procedure

To import a large decimal value from a String, the programmer may invoke the Convert procedure:

#	Argument	in	out	Type	Default	Description
1	DT	in	out	Decimal_Type	-	The decimal value to be changed
2	S	in		String	-	The input string with numeric value
3	Precision	in		Precision_Type	0	The precision of the value
4	Scale	in		Scale_Type	2	The scale of the value

The arguments *Precision* and *Scale* arguments can be omitted if you can accept the default values of 0 and 2 for the precision and scale respectively. When *Precision* is given as zero, the value has no defined precision, and may grow to whatever size is necessary to carry the result.[3]

The following example shows how to initialize a Decimal_Type from a string:

```
declare
   D : Decimal_Type;
begin
   Convert(D,"12345.6789",0,4);
```

7.7 To_String Function

To make a Decimal_Type value printable, you can call upon the To_String function:

#	Argument	in	out	Type	Default	Description
1	DT	in		Decimal_Type	-	The input decimal value
	returns			String		String representation of the value

The To_String will return the string "NULL" if the value is in the "Not a Number" state.

The following example shows how to use it in a Print_Line call:

```
declare
   D : Decimal_Type;
begin
   ...
   Put_Line("D := " & To_String(D));
```

[3]The source code indicates that the maximum precision is approximately 1,000 decimal digits.

7.8 Constrain Function

Sometimes it is desireable to constrain a result to a particular precision and scale, while watching for overflows. The Constrain function takes an input value, and returns a new value with the values constrained to the given precision and scale:

#	Argument	in	out	Type	Default	Description
1	DT	in		Decimal_Type	-	The input decimal value
2	Precision	in		Precision_Type	-	The precision for the returned value
3	Scale	in		Scale_Type	-	The scale for the returned value
	returns			Decimal_Type		The constrained value

The returned value is rounded (if necessary) to accomodate the *Scale* argument. The result must fit within the precision given by the *Precision* argument. The following example illustrates how the function is used:

```
declare
   A : Decimal_Type;
   B : Decimal_Type;
begin
   A := ...some calculation...;
   B := Constrain(A,10,2);  -- Precision 10, Scale 2
```

7.9 Expression Operations

The Decimal_Type values can be both assigned and computed with the normal set of operators:

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
unary -	Negate
=	Equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

The following code fragment shows some Decimal_Type expressions at work:

```
declare
   Watts_per_Hp : Decimal_Type;
   Watts :        Decimal_Type;
   Volts :        Decimal_Type;
   Amperes :      Decimal_Type;
```

```
   Hp :           Decimal_Type;
begin
   Convert(Watts_per_Hp,"745.577",0,3); -- Watts / HP
   Convert(Volts,"120.0",0,1);
   Convert(Amperes,"7.0",0,1);
   Watts := Volts * Amperes;       -- # of Watts
   Hp := Watts / Watts_per_Hp;     -- # of Horsepower
```

7.10 Minimum and Maximum Values

The Min_Value and Max_Value functions are available to the programmer to conveniently return the minimum or maximum value of a pair, respectively. These functions share the following common calling signature:

#	Argument	in	out	Type	Default	Description
1	Left	in		Decimal_Type	-	The left input decimal value
2	Right	in		Decimal_Type	-	The right input decimal value
	returns			Decimal_Type		The min/max value

The following example illustrates its use:

```
declare
   A, B :     Decimal_Type;
   Smallest : Decimal_Type;
begin
   Smallest := Min_Value(A,B);
```

7.11 Abs_Value, Sign, Ceil and Floor Functions

The functions in the following table are documented in this section:

Function Name	Description
Abs_Value	Absolute Value
Sign	Sign of the value (+1 or -1)
Ceil	Ceiling value
Floor	Floor value

These functions all share the following calling requirements:

#	Argument	in	out	Type	Default	Description
1	DT	in		Decimal_Type	-	The input decimal value
	returns			Decimal_Type		The result value

The following example illustrates its use:

```
declare
   A, B :    Decimal_Type;
   Pos_Val : Decimal_Type;
begin
   Pos_Val := Abs_Value(A,B);
```

7.12 Sqrt, Exp, Ln and Log10 Functions

The functions in the following table are documented in this section:

Function Name	Description
Sqrt	$\sqrt{x}$
Exp	e^{x}
Ln	$\ln x$
Log10	$\log_{10} x$

These functions all share the following calling requirements:

#	Argument	in	out	Type	Default	Description
1	X	in		Decimal_Type	-	The input decimal value
	returns			Decimal_Type		The result value

The following example illustrates its use:

```
declare
   X :   Decimal_Type;
   L10 : Decimal_Type;
begin
   L10 := Log10(X);
```

7.13 The Log Function

The Log function permits the caller to evaluate a logrithm $\log_{\text{base}} x$. It has the following calling requirements:

#	Argument	in	out	Type	Default	Description
1	X	in		Decimal_Type	-	The input decimal value
2	Base	in		Decimal_Type	-	The input number base
	returns			Decimal_Type		$\log_{\text{base}} x$

The following example illustrates its use:

```
declare
   X :    Decimal_Type;
   Base : Decimal_Type;
   L :    Decimal_Type;
begin
   L := Log(X,Base);
```

7.14 The Power Function

The Power function permits the caller to evaluate the expression x^y. The function accepts the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	X	in		Decimal_Type	-	x
2	Y	in		Decimal_Type	-	y
	returns			Decimal_Type		x^y

The following example assigns to *P*, the value x^y.

```
declare
   X : Decimal_Type;
   Y : Decimal_Type;
   P : Decimal_Type;
begin
   P := Power(X,Y);
```

7.15 The Round and Trunc Functions

To round or truncate a Decimal_Type value, the application designer may call the Round and Trunc functions respectively. They both accept the following calling arguments:

#	Argument	in	out	Type	Default	Description
1	DT	in		Decimal_Type	-	The input value
2	Scale	in		Scale_Type	-	The number of decimal places
	returns			Decimal_Type		Rounded or truncated value

The following example assigns to *R*, the rounded value of *X*, to 2 decimal places:

```
declare
   X : Decimal_Type;
   R : Decimal_Type;
begin
   R := Round(X,2);
```

7.16 Builtin Decimal_Type Constants

The builtin decimal constants are defined as the following functions:

Function Name	Value
Zero	0.0
One	1.0
Two	2.0
Ten	10.0
NaN	Not a Number (NULL)

The NaN function can be used to put a value into a "Not a Number" state. This doubles as setting the value to NULL, for SQL query use.

7.17 Using Decimal_Types with Query_Type

Creating SQL queries using the Decimal_Type and retrieving values from SQL queries has been made easy for the application programmer. The NaN state is used to represent a NULL value. This eliminates the need for the application programmer to define indicator values.

The Append procedure allows the programmer to build SQL queries with Decimal_Type values. The Value function, permits the programmer to retrieve a column value into a Decimal_Type variable (with or without a NULL value).

7.17.1 Using Decimal_Type with Append

The Append procedure has the following calling signature:

#	Argument	in	out	Type	Default	Description
1	Query	in	out	Query_Type	-	SQL Query object
2	DT	in		Decimal_Type'Class	-	The value to encode
3	After	in		String	""	Additional text to append

The following example illustrates:

7.17.2 Fetching Decimal_Type Values

A decimal value may be retrieved from a SQL query, using the Value function:

#	Argument	in	out	Type	Default	Description
1	Query	in		Query_Type	-	The SQL query object
2	CX	in		Column_Index_Type	-	The column index
	returns			Decimal_Type		The Decimal_Type result

If the returned value for the column is NULL, the value returned will be in the NaN state. The following example illustrates how to apply the Value function:

```
declare
   C : Connection_Type;
   Q : Query_Type;
   D : Decimal_Type := NaN;
begin
   ...
   Prepare(Q,”SELECT QTY, ...”);
   Append_Line(“FROM ORDERS”);
   Execute(Q,C);
   while not End_of_Query loop
      Fetch(Q);
      D := Value(Q,1);  -- Fetch Decimal_Type
   end loop;
   Clear(Q);
```

Chapter 8

Generic Database Programming

With APQ 2.0, the support of the MySQL database becomes available. The future may allow APQ to support even more database technologies. With this in mind, it becomes very desireable in some circumstances to write applications in a database neutral way. Within this documentation, we will use the term “Generic Database Programming” to describe this strategy.

This chapter is about programming for databases generically. Given that APQ is built using object oriented techniques (tagged Ada95 records), it should be possible to leverage this in a way to write application procedures once, and enjoy the flexibility of choosing or changing the database technology used later. Section 3.3.6 identified one aspect of generic database processing.

8.1 Generic Connections

For most routine database work, the only object that needs to be defined up front, is the database connection. In APQ version 2.0, there are only two concrete choices for this:

1. APQ.PostgreSQL.Client.Connection_Type
2. APQ.MySQL.Client.Connection_Type

Once the high level layer of the application chooses one of these connection objects, and establishes a connection with the database, the connection object may be passed around as parameters to procedures. The recommended generic way to do this, is to pass the connection as a Root_Connection_Type’Class parameter. See the following example:

```
procedure MyApp(C : in out APQ.Root_Connection_Type'Class) is
begin
   ...
```

The classwide parameter then permits any database connection object to be passed as a parameter. The classwide attribute causes all operations performed upon that object to

be dispatching calls (by default). By dispatching on the object's primitives, you ensure that database specific operations are carried out according to the type of database being used.

8.2 Database Specific Code

Due to the wide differences that sometimes exist between database engines, it is sometimes necessary to take different course of action, depending upon the database being used. For example, PostgreSQL allows a varchar(256) column to be defined, where MySQL is limited to varchar(255) instead.[1]

To determine the database being used, use the **Engine_Of** predicate function. This primitive exists on both the Root_Connection_Type and Root_Query_Type objects. Sections 2.8.2 and 3.3.7 describe functions and examples for this purpose.

Obviously, if you are only given a Root_Connection_Type'Class connection argument to use, you cannot know in advance which Query_Type object to use. Make use of the **New_Query** factory (See section 2.8.3) or use cloning (section 2.8.4) if you have an existing Query_Type object available.

8.2.1 Row ID Values

When designing a new system, it is important to plan for the use of Row ID values. MySQL does not support them at all, while PostgreSQL encourages their use.[2] MySQL encourages the use of serial values instead, which is a good practice. For generic database programming you must plan for these differences to reduce the amount of specialized code. For more information about obtaining row ID values, see the section 3.3.4, and section 3.3.4 for portability notes.

Additionally, the assumptions about what constitutes a null row ID must be scrutinized. Since different databases use different values to represent "no row", you should make careful use of the APQ **Null_Oid** function, rather than depend upon a particular constant. See section 3.7.2 for information about that.

8.3 Data Types

When writing generic database code it is important to choose your datatypes very carefully. One example where this is important is when using a PostgreSQL time zone type (APQ_Timezone). While a time zone variable can be used in MySQL specific code, it should be emphasized that MySQL does not support time zone values within a TIMESTAMP database column type.

A similar problem exists with PostgreSQL bit string types (APQ_Bitstring). MySQL does not support them. So if you want to write generically, stick to simple data types in your application.

[1] MySQL requires you declare the type as TEXT if you need more than 255 characters.
[2] For example, a blob is identified by a Oid value, which is basically a row ID.

When you must make use of special database types, be prepared to specialize the code somewhat, depending upon the database being used. Obviously, a designer will want to minimize this as much as possible.

8.3.1 Column Types

Another area that is important to consider is the database column types that are chosen for tables. Comparing the table in sections 1.5.1 and 1.5.2, the reader can see that most columns can be declared in SQL using the same column type declarations. However, there are some important differences. For example, a SERIAL column in PostgreSQL must be declared as an INTEGER type when using MySQL. For most applications, this is not too much of an issue, because applications usually don't create and drop tables. However, this does sometimes occur with temporary tables.

As noted in the prior section, MySQL also does not directly support some data types such as the time zone value within a DATETIME type. If time zones must be supported, the writer may simply add a time zone column declared in SQL as a SMALLINT value, and work with the time zone separately.

8.4 Pulling it All Together

This section will examine a fairly trivial example of a generic database procedure. It will make one exception for MySQL, so that the reader will know how to work with database engine differences. Ideally, you would want to avoid these differences, wherever possible.

The example is a real world example. A procedure is required to fetch the most recent stock price available on file, for a given security (by ticker symbol). While there may be a price for the security on the given day, if there is not one listed, the procedure is expected to fall back to the most recent price available. The table being consulted, is defined as follows:

```
CREATE TABLE PRICE_HIST (
    SECURITY   CHAR(10) NOT NULL,
    PRICE_DATE DATE NOT NULL,
    PRICE      REAL NOT NULL,
    PRIMARY KEY(SECURITY,PRICE_DATE)
);
```

Here is the package spec for the Prices module, which makes the procedure Last_Price available for use:

```
with APQ;
use APQ;

package Prices is

   procedure Last_Price(
       C :       in out Root_Connection_Type'Class;
      Security : in     String;
```

```
      Price :       out APQ_Double
   );

end Prices;
```

Given any database connection, and a ticker symbol provided in argument Security, the procedure Last_Price is expected to lookup the most recent stock price in the PRICE_HIST table and return that price in the Price argument. Here is the body of the package, written to work with any database:

```
package body Prices is
   procedure Last_Price(
      C :        in out Root_Connection_Type'Class;
      Security : in     String;
      Price :       out APQ_Double
   ) is
      function Value is new Float_Value(APQ_Double);

      Q : Root_Query_Type'Class := New_Query(C);
   begin

      Prepare(Q,     "SELECT SECURITY,PRICE_DATE,PRICE");
      Append_Line(Q, "FROM PRICE_HIST");
      Append(Q,      "WHERE SECURITY = ");
      Append_Quoted(Q,C,Security,Line_Feed);
      Append_Line(Q, "ORDER BY SECURITY,PRICE_DATE DESC");

      if Engine_Of(C) = Engine_MySQL then
         Append_Line(Q,"LIMIT 1");
      end if;

      Execute(Q,C);

      begin
         Fetch(Q);
      exception
          when No_Tuple =>
             raise;    -- Indicates no price
      end;

      Price := Value(Q,3);

   end Last_Price;

end Prices;
```

A few notes are necessary: The spec already with's and uses the package APQ. So it is not repeated in the body of the package. The Last_Price procedure takes a Root_Connection_Type'Class connection object type, so it can be supplied with a MySQL or PostgreSQL database connection.

The New_Query factory function is used to create the correct Query_Type object necessary to form and execute the query. The Prepare, Append_line, Append_Quoted calls build up an SQL query, and then the type of the database is queried by calling

Engine_Of. If the database being used is a MySQL database, the query is optimized[3] so that only one row is returned by use of the extended SQL "LIMIT 1" clause.

Note that the "ORDER BY" clause requires that the sort order be descending (most recent dates first). Given that the "ORDER BY" clause specifies indexed columns, this retrieval should be quick since a reverse index retrieval is possible (if the database cannot do this, you should create a new index or fix the primary key to be descending).

Since we are only interested in the most recent price (ie. one price), only one Fetch call is made. This is not a problem for PostgreSQL, but it is for MySQL if there there are more than one rows of result (see section 3.5.1 to find out why). If the database is MySQL the problem is addressed by adding to the query a MySQL extended clause "LIMIT 1", to limit the results to one row.

The price is then fetched from column 3 (PRICE) and the procedure returns. If no rows are returned, we simply raise the APQ.No_Tuple exception here to keep the example simple. A finished application would either declare a proper application specific exception, or handle the problem with an indicator.

The important thing to recognize here is that there is nothing specific to the type of database being used in this Prices package, except for the MySQL work-around. The only APQ package being used is APQ, and the root types for connection and query types.

Here is a PostgreSQL main program:

```
with Ada.Text_IO;
with APQ.PostgreSQL.Client;
with Prices;

use APQ, Prices, APQ.PostgreSQL.Client, Ada.Text_IO;

procedure Price_PG is
   C : Connection_Type;
   P : APQ_Double;
begin

   Set_DB_Name(C,"investments");
   Connect(C);

   Last_Price(C,"RHAT",P);
   Put_Line("RHAT $" & APQ_Double'Image(P));

   Disconnect(C);

end Price_PG;
```

Please notice the following points about the main program:

1. The database specific package is with'ed as APQ.PostgreSQL.Client here to choose the database connection being used.

2. The Connection_Type object is declared in APQ.PostgreSQL.Client and derives from APQ.Root_Connection_Type.

[3] One could also argue that it is "fixed" here, since MySQL insists that all row results of a query be fetched.

3. The database is chosen and a connection is established.

4. The Last_Price procedure is called, providing only the connection and the security's ticker symbol that the price is being sought for.

5. The returned price P is then printed (crudely)

6. The application disconnects from the server and exits.

The main program for MySQL use, is virtually identical:

```
with Ada.Text_IO;
with APQ.MySQL.Client;
with Prices;

use APQ, Prices, APQ.MySQL.Client, Ada.Text_IO;

procedure Price_My is
   C : Connection_Type;
   P : APQ_Double;
begin

   Set_DB_Name(C,"investments");
   Connect(C);

   Last_Price(C,"RHAT",P);
   Put_Line("RHAT $" & APQ_Double'Image(P));

   Disconnect(C);

end Price_My;
```

The only difference between this MySQL main program and the prior PostgreSQL main program, is the name of the package used (APQ.MySQL.Client). APQ truly is the closest thing to database independance!

8.5 Miscellaneous Portability Issues

There are a number of other database portability issues that should be born in mind. An incomplete list has begun in this document below:

- temporary tables creation
- SELECT ... INTO TABLE ...

This list will likely grow as the author and the APQ community applies APQ to generic database programming.[4]

[4]Contributions are welcome.

8.5.1 Temporary Tables

Many databases allow the SQL programmer to create a temporary table prior to its use in the application. This temporary table is only visible to the user of the established database connection. When the database connection is closed or disconnected, the temporary table is automatically discarded and its space recycled by the database engine.

The difficulty that a generic database programmer needs to be aware of is that some databases work differently. Normally, an application would perform something like the following to create a temporary table in the database session that required it:

```
CREATE TEMP TABLE INTERMED_RESULTS (
   SECURITY  CHAR(10) NOT NULL,
   HOLDINGS  BIGINT NOT NULL
);
```

Subsequent to the successful creation of this temporary table, the application would populate it and use it as necessary. The application may even create indexes on the table after the table is populated, to help performance in later stages of the table's use.

While this works for PostgreSQL and MySQL, the generic programmer should be aware that this will not work on an ORACLE database (when APQ gets there some day). ORACLE permits the same syntax, but operates differently: ORACLE only permits one CREATE TEMP TABLE operation to be performed, in the same way that a permanent table is created. Once created, the user implicitly gets access to the table upon demand. Upon the first reference to the temporary table in a particular session, you get a temporary table created, which is empty. You then populate and use that temporary table without ever having to create it within that particular session. When the session is over, the table's contents are discarded.

So how do you plan for this? If the **Engine_Of()** function indicates **Engine_ORACLE**, you must *not* create the temporary table during your database session. This will be done when the permanent tables are created.[5] For other database engines, you should create the temporary table when you are about to use them in your application.

Indexes on Temporary Tables

There is an additional piece of advice to consider when creating indexes for temporary tables. For performance reasons, it is often best to populate the temp table with no indexes created. After the table has been populated, indexes can then be efficiently added and dropped as the needs arise.

In the ORACLE case, these indexes are likely to be predefined as is the declaration of the temporary table itself. So when creating indexes for temporary tables, you should also probably test for ORACLE in the generic code. When using ORACLE, you probably do *not* want to create or drop the index, as it will probably affect all users of that temporary table definition.[6]

[5] In many respects, this is perhaps the best time to declare and plan for a temporary table.

[6] The author has not verified this point, and the reader is encouraged to do so.

8.5.2 SELECT ... INTO TABLE

A number of database engines support a SQL syntax along the lines of:

```
SELECT *
FROM MY_TABLE
WHERE ...
INTO TEMP TABLE TEMP123
```

The above SQL code performs the usual SELECT, but places its results into a temporary table named TEMP123.[7] Alternatively, databases will also often permit:

```
SELECT *
FROM MY_TABLE
WHERE ...
INTO TABLE RESULTS
```

This query places the results into a permanent table named RESULTS (note that the keyword TEMP was dropped).

Both of these operations are very convenient for processing intermediate results. However, database engines vary in their support. Neither of these formats are supported by PostgreSQL or MySQL, but they are supported by INFORMIX.

The SQL-99 way to perform this operation is specified as follows:

```
INSERT INTO RESULTS
SELECT *
FROM MY_TABLE
WHERE ...
```

This syntax is supported by both PostgreSQL and MySQL for permanent tables.

There is no provision currently to create a temporary table on the fly with syntax shown on page 132 using PostgreSQL or MySQL. So even SQL-99 syntax won't help you there. You can only create a temporary table, and then use the INSERT INTO ... SELECT syntax after the temporary table is created. But don't forget the limitation in section 8.5.1.

[7] One application framework that the author is familiar with used a suffix of "123" to denote the name of a temporary table.

Chapter 9

Troubleshooting

There are several problems that can crop up in applications using the APQ Binding. These problems usually fall into one of the following categories:

- PostgreSQL database server “personality”
- The PostgreSQL libpq C library interface
- The APQ binding itself

The APQ Binding attempts to insulate the user as much as is practical from the libpq C library issues and the database server. However, some issues still manage to poke through. This chapter is an attempt to provide some useful advice for those people that are encountering unexpected behaviour, using the APQ Binding.

9.1 General Problems

The following subsections provide general troubleshooting help with APQ binding issues.

9.1.1 Missing Rows After Inserts

The first step in identifying whether this section applies to you or not, is to ask:

- is transaction processing being used?
- or, is a transaction pending when it shouldn’t be?

If the second bullet applies to you, then you need to correct the logic in your program (but read on to find out why).

On the other hand, if you are purposely using transactions (first case) and you are losing inserted row information, then it is likely that you are suffering from an aborted transaction. It may also represent an APQ binding bug.

The APQ Binding should prevent aborted transactions from being left unnoticed.[1] When the database server notificaties the APQ binding that an "abort state" has been entered[2], any further attempts to execute SQL queries or COMMIT WORK on that connection, will raise the Abort_State exception (see also sections 2.5.4 and 3.4).

One common reason is an application inserts row(s) into a table, and intercepts the SQL_Error exception. This exception is caught because the application writer wants to ignore the insert on a duplicate key error. The difficulty here is that the database engine will enter an "abort state" after the failed INSERT operation, regardless of how the application handles the exception. The only recourse to recovery at this point is to rollback the transaction with a call to Rollback_Work (section 3.4).

This brings up a question about the APQ Binding. Why doesn't the APQ binding raise Abort_State immediately after the failed INSERT operation within a transaction? There are two reasons:

1. The "Abort State" notice is provided to the APQ binding in the form of a callback.

2. Processing SQL_Error exception within a transaction implies an aborted transaction.

Notices are received by the APQ binding by registering a callback with the database server. As a result, it is not always possible to know about the "abort state" when it might be critical to the application. Even if the information is available, this may not always be so in future versions of PostgreSQL (timing may change).

The very fact that you've started a transaction with Begin_Work and you have encountered an SQL_Error exception should tell you that you must Rollback_Work and recover. So as the developer, you should be thinking:

$Abort\ State = Begin\ Work + SQL\ Error$

The APQ binding has been designed to avoid several SQL statements from being executed and being ignored because the database server is in the "Abort State". This is why the Execute and Commit_Work calls check for this and raise the Abort_State exception. However, the best advice is to not rely on this mechanism when programming the logic of your application.

9.1.2 Missing Time Data (Or Time is 00:00:00)

You build a query to insert or update a row with date and time information, but only the date is getting stored in the database. The time component always reads midnight (00:00:00). Or perhaps you provide a date and time stamp as part of a where clause, but it fails because only the date value is being put into the query (the time component always shows as midnight). You print out the variables using To_String and they show the correct values, as follows:

[1] If however, the Abort_State exception is never being raised, then it is possible you have a PostgreSQL porting issue. If the PostgreSQL notice message format changes, the APQ bindign code will fail to recognize the notification of an "abort state" from the database server.

[2] After a duplicate key on insert error, for example.

```
declare
   type My_Date_Type is new APQ_Timestamp;
   My_Date : My_Date_Type;
begin
   ...
   Put_Line("My_Date="'
      To_String(APQ_Timestamp(My_Date))
      & ""');
```

The above symptoms are the result of a common problem. This human error is easy to make and is due to choosing the incorrect generic procedure. Look for a generic instantiation statement like the following:

```
procedure Append
   is new Append_Date(My_Date_Type);
```

If your data type *My_Date_Type* holds time information, then it is likely that you meant to code the following instead:

```
procedure Append
   is new Append_Timestamp(My_Date_Type);
```

The error was choosing generic procedure Append_Date over the correct Append_Timestamp routine.

A similar mistake can be made choosing between Encode_Date and Encode_Timestamp generic procedures. So watch for these subtle differences!

9.1.3 Exception No_Tuple

If you are having the exception No_Tuple being raised when you don't think it should be, then check to see if the following apply:

- Are you using MySQL?
- Do you have End_of_Query function calls used?

If you do, then you need to look for code structured as:

```
while not End_of_Query(Q) loop
   Fetch(Q);
   ...
end loop;
```

This type of code works well for PostgreSQL, but may be problematic with some other databases (MySQL has a problem with this). Restructure your code to eliminate the calls to End_of_Query. Section 3.5.5 describes the problem in greater detail. Restructure the loop to something like the following:

```
loop
   begin
      Fetch(Q);
   exception
      when No_Tuple =>
         exit;
   end;
   ...
end loop;
```

9.1.4 Database Client Problems

If problems with the database engine begin to occur after a particular query, look for the following:

- Are you using MySQL?
- Are your doing a SELECT, or otherwise returning row results?
- Is your Query_Type object in Sequential_Fetch mode?
- Is your code fetching all row data?

This problem may occur with MySQL, since the client library for MySQL requires that all row result data be fetched. Failure to fetch all rows may cause a backlog in communication with the database server and cause a multitude of strange behaviour and errors, when new queries are started (with the old results still waiting to be fetched). The Query_Type object defaults to Random_Fetch mode however, so unless your code has changed the fetch mode of the object, this may not be your problem.

9.1.5 Client Performance or Memory Problems

Check to see if the following apply:

- Are you using MySQL?
- Are you doing a SELECT or otherwise returning large row sets?
- Is your Query_Type object in Random_Fetch mode? This is the default.

If the above are all true, then it is possible you have formed a query that has generated a large row set. Since the default for the Query_Type object is for Random_Fetch mode, the APQ library calls upon mysql_store_result() to fetch all of the result set into the client memory for random access. For reasonable sized sets of rows, this works well, but for large results this can be very expensive and may run your application out of memory.

Consider the PRICE_HIST table like the one discussed on page 127. For MySQL, if you fail to include the LIMIT 1 clause, and your Query_Type object uses the default Random_Fetch mode, you could easily find your application selecting the entire history

of one security into your application client memory! If you have several years of price history for that stock, your application may be destined to run out of memory the first time that query is run.

When using the MySQL database, you must consider the following:

- MySQL fetches in Random_Fetch mode must guarantee a *reasonably small result set* (use the LIMIT clause if necessary).
- MySQL fetches in Sequential_Fetch mode must fetch *all* row data

Unless otherwise noted, you probably do not need to be concerned about this. PostgreSQL for example, does not require all row data to be fetched. However, if there are ways to restrict the row set, this may improve performance for any given database engine involved, and should be considered.

9.1.6 Can't Find Existing Table Names

Some databases use caseless references to database objects, while others are case sensitive (MySQL can be either). If you are using MySQL and experiencing problems, consider adding the parameter:

```
[mysqld]
set-variable = lower_case_table_names=1
```

to your MySQL database configuration. By default, MySQL distinguishes between table names PRICE_HIST, Price_Hist, and price_hist for example.

9.1.7 Failed Transactions

You should be aware that PostgreSQL will abort a transaction if an insert is attempted that proves to be a duplicate entry (or has a duplicate key). APQ tracks this status (Abort Status) to help the application programmer debug his application.

Other databases, like MySQL do not abort the transaction if one step in the transaction fails (like a duplicate insert). For this reason, you may need to review how your application code is dealing with situations like potential duplication inserts.

This problem is mostly likely to be noticed when you have developed your application using a database like MySQL, and then migrated it to PostgreSQL.

9.2 Blob Related Problems

Blob operations on a database can be tricky to get right. The following subsections provide assistance with blob related problems.

9.3 Blob_Create and Blob_Open Fails

You have written what seems like a simple piece of code that creates a blob, and then writes some data to it. It couldn't possibly fail on paper, but it does when you run it. Or you are wondering why that Blob_Open call keeps failing, because you are certain that the OID of that blob surely does exist. These are both symptoms of the same problem!

> *"All blob operations in PostgreSQL must be performed within a transaction "*

Repeat it to yourself again.

Unless you have started a transaction on the connection that you are using, all blob operations will fail. They will only succeed within a transaction. Furthermore, make certain your application commits the changes to your blob, after they have been performed successfully. Otherwise your application may fall prey to the default actions of PostgreSQL, which may be to rollback your changes.[3]

9.4 Blob I/O Buffering Bugs Suspected

If you have good reason to believe that the APQ binding software[4] has a bug in its buffered blob I/O, you can disable blob I/O buffering. This is done by specifying a value of zero for the Buf_Size argument in the Blob_Open and/or Blob_Create calls that you are troubleshooting. Be prepared to accept a large degradation in performance when specifying unbuffered I/O this way. The performance is especially poor when array I/O is performed.

Another possibility might be that you need to call upon Blob_Flush at strategic points in your application. While the buffering algorithms used are such that you should not need to worry about this, it is worth investigating.

Note also that multiple write access to the same blob is definitely *not* supported by APQ.

9.5 Transaction Problems

The following subsections deal with transaction problems that may occur with application termination.

9.5.1 Abnormal Termination of Transactions

The APQ binding is designed to commit or rollback a transaction when the Connection_Type object finalizes. The default behavior of the Connection_Type is to ROLLBACK WORK, when the object finalizes[5]. Consequently, if your program raises an uncaught exception (perhaps Program_Error or Constraint_Error), the Connection_Type

[3] Check your PostgreSQL documentation to determine what the default transaction action is for your version of PostgreSQL database.

[4] Only APQ versions 1.2 and later have buffered blob I/O.

[5] Provided that the Connection_Type object is connected to the database at the time of finalization.

object will finalize and rollback the transaction on you.[6] If this is undesired behaviour, then check out the Set_Rollback_On_Finalize primitive in section 2.6.1.

9.5.2 Aborted Applications

The APQ binding can only perform the default COMMIT/ROLLBACK action (see section 2.6.1) if the Connection_Type object is permitted to have its Finalize primitive called. If the process under UNIX for example, is terminated with a signal (by the kill(1) command), the objects within your application may not experience a Finalize call, because normal Ada shutdown procedures were not invoked.[7] If this is the reason for your problem, then you have two courses of action:

- Commit/Rollback explicitly in the program (prior to receiving a signal)
- Avoid signalling the application
- Add signal handling capability to your Ada application, to permit an orderly application shutdown

The choice is generally up to the application designer. Whenever possible however, where it is important, the application should perform its own explicit commit or rollback operation.

9.6 SQL Problems

If you are experiencing SQL problems that you don't understand, the quickest way to inspect what is really going on is to use the APQ trace facility. The documentation for the SQL trace facility is given in section 2.7.

9.6.1 Tracing SQL

Where your Connection_Type connects to the database, add a call to Open_DB_Trace as follows:

```
declare
   C : Connection_Type;
begin
   ...
   Connect(C);
   Open_DB_Trace(C,"trace_file.txt",Trace_APQ);
```

Without adding another line of code, every SQL interaction will be captured to trace_file.txt, which you can inspect when the application completes.

[6] Unless you have changed the default setting for the object.

[7] There are Ada95 ways to deal with UNIX signals, which permit an orderly Ada shutdown of your application.

9.6.2 Too Much Trace Output

If your application performs so many SQL operations that the trace file becomes too large, then disable the tracing until you get to a strategic point in the program:

```
declare
   C : Connection_Type;
begin
   ...
   Connect(C);
   Open_DB_Trace(C,"trace_file.txt",Trace_APQ);
   Set_Trace(C,False);  -- Disable trace for now..
   ...
   Set_Trace(C,True);   -- Start tracing now
```

The overhead of the Set_Trace primitive is light, unless you have selected Trace_libpq or Trace_Full (these add the overhead of invoking libpq functions PQtrace() and PQuntrace()). Light overhead permits you to use Set_Trace within a loop for example, to gather only the information you need.

9.6.3 Captured SQL Looks OK

If the SQL code captured in section 9.6 looks OK, but the database engine is still reporting a problem, then try the following:

1. Create a capture file using Trace_APQ.
2. Edit out the portion of the SQL queries in the capture file that you are having difficulty with.
3. Use the PostgreSQL psql command and replay the extracted problem SQL queries.
4. Edit SQL query and repeat step #3 as necessary.

This allows you to experiment with the SQL text as your applicaton created it. Once you achieve success with psql, then you can go back and correct your application to form the query correctly.

9.6.4 You Want to Report a Problem to PostgreSQL

If you want to report a trace file in terms that the PostgreSQL people understand, simply choose the Trace_libpq trace mode when creating a trace file. Then send them the trace file with a description of the problem.

9.6.5 Missing Trace Information

The trace information is collected at the Connection_Type object level. Check to see if you have more than one Connection_Type object involved. If so, make sure you set the appropriate trace settings on the connections that you want to collect trace information for.

Note also, that once a Connection_Type object finalizes, its trace file is closed and its trace state is lost.

9.7 Connection Related Problems

If you are experiencing trouble establishing a connection to the database itself, then there are a number of environment related issues. A number of environment variables affect a PostgreSQL database connection:[8]

PGHOST Host name of the database server

PGPORT IP port number or UNIX socket pathname of the database server

PGDATABASE Database name within the database server

PGUSER Database user name

PGPASSWORD Database password

PGREALM Kerberos realm for the database server

PGOPTIONS Database server options

Any of these connection mode parameters that are not configured in the application are defaulted to the ones defined by the above environment variables. If you are experiencing trouble, make certain that your variables are *exported*. In many shells, like the Bourne and Korn shells,[9] this is done as follows (for the PGHOST variable):

```
export PGHOST
```

Some shells, like the Korn shell and the GNU bash shell, allow multiple variable names to be listed at once:

```
export PGHOST PGPORT PGDATABASE PGUSER
```

Once you have the environment configured correctly, you should be able to access the database with the PostgreSQL psql command. If the psql[10] command still fails, then you may need to revisit your environment variable settings or possibly even the database server configuration.[11]

9.7.1 Connection Cloning Problems

If the original Connection_Type object connects OK, but the Connect clone call fails, then it could be that the network has gone bad since the original connection was made. If exceptions other than Not_Connected are being raised, then check that:

- Make sure the parameters are in the correct order in the Connect call.
- Make certain that the connected object, is indeed connected.
- Make certain that the new object is not already connected.

[8] Check your PostgreSQL documentation for the final word on this subject.
[9] I won't encourage anyone here to use the csh.
[10] Do a "man psql" for details about the psql client command.
[11] Check the security aspects of your connection first.

9.7.2 Connection Tracing

Problem: Your first Connection_Type object is tracing to a file successfully, but the cloned Connection_Type object is not.

Reason: Cloned connections do not have the trace file parameters cloned. This was a compromise to make APQ more portable to other platforms that may not share files well.

Solution: Configure the cloned connection to trace to a file separate from the original connection.

Chapter 10

Appendix A - PostgreSQL Credits

PostgreSQL Decimal C Sources

PostgreSQL Database Management System (formerly known as Postgres, then as Postgres95)

- Portions Copyright (c) 1996-2001, The PostgreSQL Global Development Group
- Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Modification Notice

The numeric C routines required extensive interface modifications for use in this APQ Binding. These modified C sources were extracted from the PostgreSQL server project. No guarantee is made with regard to the quality of these modifications.

Contributor Notice

In no event shall the author or contributors to the APQ Binding be liable to any party for any cause that the modified PostgreSQL software may cause or contribute to.

Chapter 11

Appendix B - APQ License

Scope of the APQ Binding License

The "APQ Binding" license covers those software modules not provided by or extracted from the PostgreSQL database software (such as the Decimal C source modules).

APQ Binding License

The APQ Binding is covered under a dual-license arrangement. This should be reflected in the file name COPYING. The following two licenses are available:

1. The Ada Community License (ACL).
2. The GNU Public License 2 (GPL2)

One or the other license must be chosen to cover the APQ sources being used. For the ACL license details, see `http://www.adapower.com/booch/ACL/index.html`, file ACL.txt or Appendix C. For details about the GPL license, see Appendix D or the file GPL.txt that was included with your software.

Chapter 12

Appendix C - Ada Community License

The Ada Community License

Permission to redistribute in unmodified form is granted, all other rights reserved. This is a modification of the Perl Artistic License, (c) 1989-1991, Larry Wall

Preamble

The intent of this document is to state the conditions under which the Ada library may be copied, such that the Copyright Holder maintains some semblance of artistic control over its development, while giving Ada users the right to use and distribute the Ada library in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

"Library" refers to the collection of Ada source files distributed by the " Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version" refers to such a library if it has "Standard Version" not been modified, or has been modified as specified below.

"Copyright Holder" is whoever is named in the copyright or copyrights for the Ada library.

"You" is you, if you're thinking about copying or distributing this library.

"Reasonable Copying Fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required

to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

Provisions

1. You may make and give away verbatim copies of the source form of the Standard Version of this Ada library without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A library modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Ada library in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

   (a) Place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as The Public Ada Library, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Ada library.

   (b) Use the modified Ada library only within your corporation or organization.

   (c) Rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.

   (d) Make other distribution arrangements with the Copyright Holder.

4. You may distribute the programs of this Ada library in object code or executable form, provided that you do at least ONE of the following:

   (a) Distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.

   (b) Accompany the distribution with the machine-readable source of the Ada library with your modifications. Accompany any non-standard executables with their

   (c) corresponding Standard Version executables, giving the non-standard executables non-standard names, and clearly documenting the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.

(d) Make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Ada library. You may charge any fee you choose for support of this Ada library. You may not charge a fee for this Ada library itself. However, you may distribute this Ada library in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Ada library as a product of your own.

6. The scripts and library files supplied as input to or produced as output from the programs of this Ada library do not automatically fall under the copyright of this Ada library, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Ada library.

7. System-level subroutines supplied by you and linked into this Ada library in order to emulate the functionality defined by this Ada library shall not be considered part of this Ada library, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the library in any way that would cause it to fail the regression tests for the library.

8. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

9. THIS ADA LIBRARY IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

How to Apply These Terms to Your New Libraries

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found. Also, be sure to add information on how to contact you by electronic and paper mail.

Copyright (C)

This program is free software; you can redistribute it and/or modify it under the terms of the "Ada Community License" which comes with this Library. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the Ada Community License for more details. You should have received a copy of the Ada Community License with this library, in the file named "Ada Community License" or "ACL". If not, contact the author of this library for a copy.

Chapter 13

Appendix D - GNU Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and

2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty;

keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code. 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License. 8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an

explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR

A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Found
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A P
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place,
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and yo
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Chapter 14

Appendix E - Credits

This appendix documents the contributors to the “APQ Binding” that are separate from the PostgreSQL project.

Author


```
Warren W. Gay VE3WWG
29 Glen Park Road,
St. Catharines, Ontario
Canada L2N 3E3

ve3wwg@cogeco.ca
```


Contributions

Source Code Modifications

There are no other contributors at the present time.

Bug Reports

- There appears to be a problem with win32 releases using PostgreSQL 7.2.1, where specifying the database server by IP # creates a problem. The problem is believed to be in libpq.dll.

Suggestions

None yet.

Bug Report/Fix Contributions

- Charles Darcy <charlie@mullum.com.au>, November 23, 2002

Chapter 15

Appendix F - History

APQ 1.0

First released August 3, 2002, under the Ada Community License (ACL).

APQ 1.1

Second release August 4, 2002, but now under the dual ACL and GPL2[1] license. This license change was suggested by Florian Weimer.

APQ 1.2

- Added function End_Of_Blob to streamline sequential processing.
- Added buffered blob I/O for higher performance.
- Added procedure Blob_Flush to force unwritten blob data to the database server.
- Blob_Create has new optional Buf_Size argument.
- Blob_Open has new optional Buf_Size argument.
- Fixed Blob_Create to release the created blob, if the blob cannot be opened. This most often happens when the caller is attempting to create a blob, outside of a transaction.

APQ 1.3

- Removed some debug Put_Line statements that should have been removed in 1.2.

[1] GNU Public License 2

- Added a few pragma Inline statements to the spec PostgreSQL.Client

APQ 1.4

- Added Generic_Command_Oid for strong PG_Oid type use
- Added Generic_Blob_Open for strong PG_Oid type use
- Added Generic_Blob_Oid function for strong PG_Oid type use
- Added Generic_Blob_Unlink for strong PG_Oid type use
- Added Generic_Blob_Import and Generic_Blob_Export for strong PG_Oid type use

APQ 1.5

- Bug fix: Append_Time, Append_Date and Append_Timestamp now emit the surrounding single quote characters around the value to satisfy the SQL syntax required by the database server.
- Troubleshooting help added to this manual for “Missing Time Data (Or Time is 00:00:00)”.

APQ 1.6

- Added Set_Rollback_On_Finalize controlling primitive for Connection_Type.
- Added Will_Rollback_On_Finalize function for Connection_Type for inquiry.
- Expanded manual and added transaction problem help to the Troubleshooting chapter.

APQ 1.7

- Open_DB_Trace and Close_DB_Trace procedures added.
- Set_Trace procedure added to enable and disable tracing.
- Is_Trace function added to query the tracing state.

APQ 1.8

- Connection information functions like Host_Name and Port were added.
- A connection cloning primitive was added.

APQ 1.9

- A compiler work-around was provided. Some versions of GNAT would not compile the APQ source code, because certain instantiations of Ada.Text_IO.Integer_IO were producing duplicate symbol errors in the assembler phase of the compile. This problem was absent in gnat 3.13p compiles, but have shown up in gcc-3.1.1 and probably in gnat 3.14p and later releases. The instantiation names INTIO were made unique within the source code to avoid this problem.

APQ 1.91

- The Connect call now performs an automatic "SET DATESTYLE TO ISO" command prior to returning from a successful connect. This is necessary to guarantee that ISO date format is returned from the database engine and recognized from the engine. This guarantees that APQ correctly handles dates, even when the user has specified a PGDATESTYLE environment variable value that is different than ISO.

- Fixed bug in Host_Name function. It was returning a null string when a host name was set in the Connection_Type object.

- Win32 apq-1.91 source release (subdirectory win32) and apq-1.91-win32-2.7.1 binary release created.

APQ 1.92

- Fixed bug for floating point and fixed point types (was rounding the value to the nearest integer, due to the fact that the Ada.Text_IO.Float_IO.Put call was receiving the argument Aft => 0). Omitting the Aft parameter causes the value to be formatted as required for the SQL floating/fixed point type. The bug was reported by Charles Darcy <charlie@mullum.com.au>.

APQ 1.93

- Modified the Ada95 package hierarchy to insert a top level package named APQ. Hence package PostgreSQL now becomes APQ.PostgreSQL. This is an interim release, which will pave the way to future support of other database products such as MySQL.

APQ 2.0

- MySQL support added. This was made possible by the package restructuring done in the interim release APQ 1.93.

- Generic database programming support added. Special generic services like Engine_Of, New_Query etc. were added to make this possible. A heavy reliance is made upon object inheritance and polymorphism to make this work.
- A new example subdirectory eg2 was added. The programs in this subdirectory show the original example program, but done in a database generic way. The test program can be compiled and run for both PostgreSQL and MySQL databases.
- PG_Oid is now named APQ_Row_ID and is 64 bits unsigned integer.
- Null_Oid() function added for generic database support (and much more).
- Engine_Of() function added for generic database support.
- Exception “Failed” was added to handle some general failures.
- Fetch_Mode() and Set_Fetch_Mode() were added to accommodate MySQL limitations.
- Documentation went through some restructuring to accomodate two databases, and their differences.

APQ 2.1

- This was the win32 port.
- See win32.pdf for instructions for building APQ on a win32 platform.
- win32_test.adb program was added to the distribution to allow testing of APQ in the windows environment.