Part I: Introduction

1. Overview

The Generic Database Interface (GDI) is a common interface to multiple databases, providing two key capabilities:

1. *Database Access*
   An application executes a database query with the same GDI call whether the target database is ORACLE, POSTGRES, or SYBASE.
2. *Data Management*
   Data to and from the database are managed in the native format of the application, making it possible to provide a seamless integration of application and database.

The GDI model consists of the interfaces depicted in Figure 1. Interfaces may be added to increase functionality without having to modify lower level interfaces.

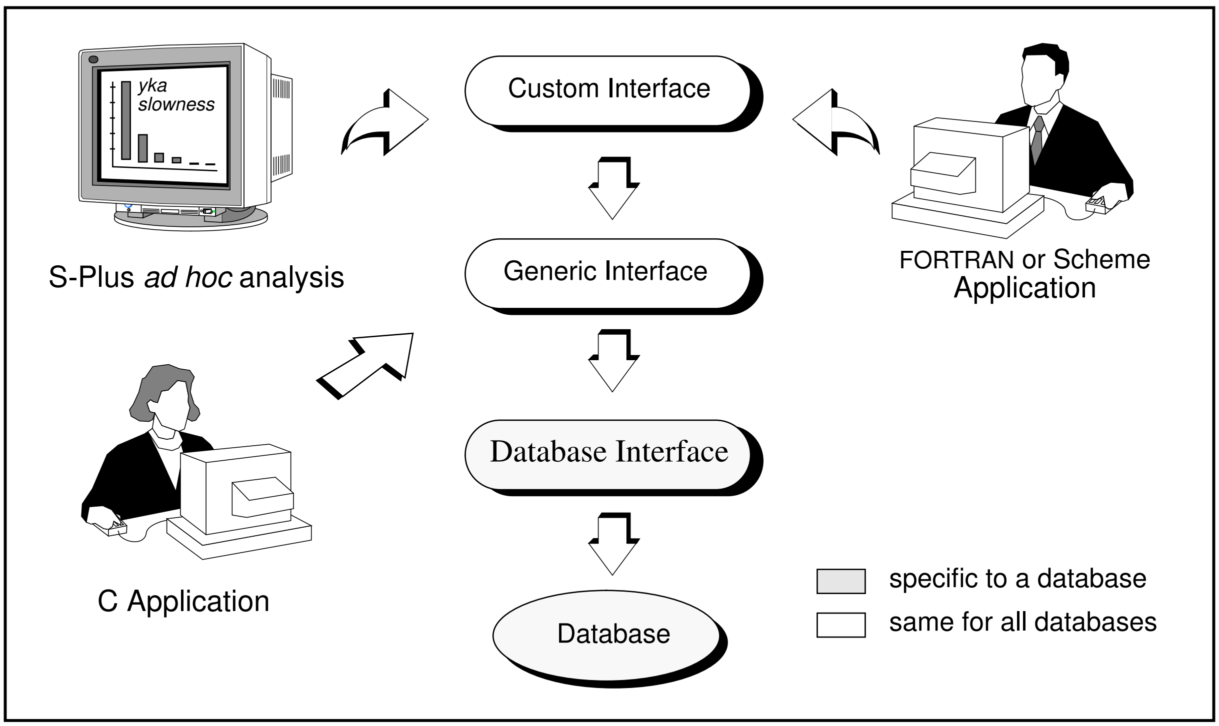

FIGURE 1. Generic Database Interface (GDI) Model

Working from the bottom of Figure 1 to the top, the GDI consists of

- *Database Interface*
  Manages interaction with the target database.
- *Generic Interface*
  Provides a generic API (application program interface) for a C application to access any database and manage data.
- *Custom Interface*
  Supports programming languages such as FORTRAN and Scheme, and third-party products such as S-Plus.

1.1 Intended Audience

The GDI targets two types of users: the end-user and the application developer.

The end-user interactively accesses the database with a program created by an application developer or a third party tool such as S-PLUS. End-users want a "hot link" between the application and the target database so they can concentrate on research and analysis. They do not want to be sidetracked by having to manually transfer data, not even with the aid of data migration tools.

The application developer writes programs that require database access. Application developers want a consistent interface between the application and the target database so they can concentrate on a specific area of programming expertise, whether it be the design of sophisticated user interfaces or complex scientific programs. They do not want to be sidetracked by having to learn how to access each database.

Neither user wants to become an expert for each database accessed. Both want application and database to be transparently integrated. The GDI achieves that transparent integration.

This manual describes what each user must know to submit queries to a database and manage data. The user needs to know:

- The database query language, which is a topic beyond the scope of this document. Appendix A lists a few SQL references. POSTGRES documentation is available *via* anonymous ftp from *postgres.berkeley.edu*.
- How to use the generic functions that execute queries and manage data. This is the topic of this manual.

The user does not need to know:

- Database vendor-specific implementation of Embedded SQL and/or the call interface.
- Database vendor-specific data dictionary structure.
- Database vendor-specific error handling.
- Application-specific and database-specific data formats.
- Internal GDI data structures.

1.2 Document Organization

PART I introduces a high-level view of the GDI. Section 1 (this section), describes the GDI model. Section 2 describes the GDI architecture.

PART II introduces GDI routines to the application developer. Section 3 discusses naming conventions, sample programs, and known problems. Section 4 discusses managing database queries. Section 5 describes data management. Sections 6 and 7 discuss error handling and transaction management, respectively.

PART III introduces custom interfaces to the end-user. Section 8 contains an S-PLUS tutorial.

PART IV contains UNIX Section 1 man pages for GDI tools and Section 3 man pages for GDI routines. The most current man pages will always be available on-line.

PART V contains appendices. Appendix A is a bibliography of SQL language references. Appendix B is a description of GDI data types.

1.3 Related Documents

The *Generic Database Interface Requirements* document describes basic GDI requirements.

The *Generic Database Interface Design* document describes each component of the GDI.

1.4 User Feedback

The GDI development team welcomes comments. All bug reports and suggestions for improvement should be sent to Jean Anderson (*jean@gso.saic.com*).

2. Architecture

Section 1 presented a high level view of the GDI. This section describes the key components of the GDI architecture:

- *Basic Services*: Database access routines.
- *Database Connector (dbConn):* Manages database queries.
- *Database Object (dbObj):* Manages data to and from the database.

Figure 2 depicts how an application uses the dbConn and dbObj to access a database. All queries are executed on the dbConn that was established when the application connected to the database. This is similar to a C program using a FILE pointer for reads and writes to a file opened with *fopen()*. If a query returns data, the GDI returns a pointer to the dbObj containing the data. If an application needs to insert data into the database, it can create a dbObj and populate it with the data to be inserted.

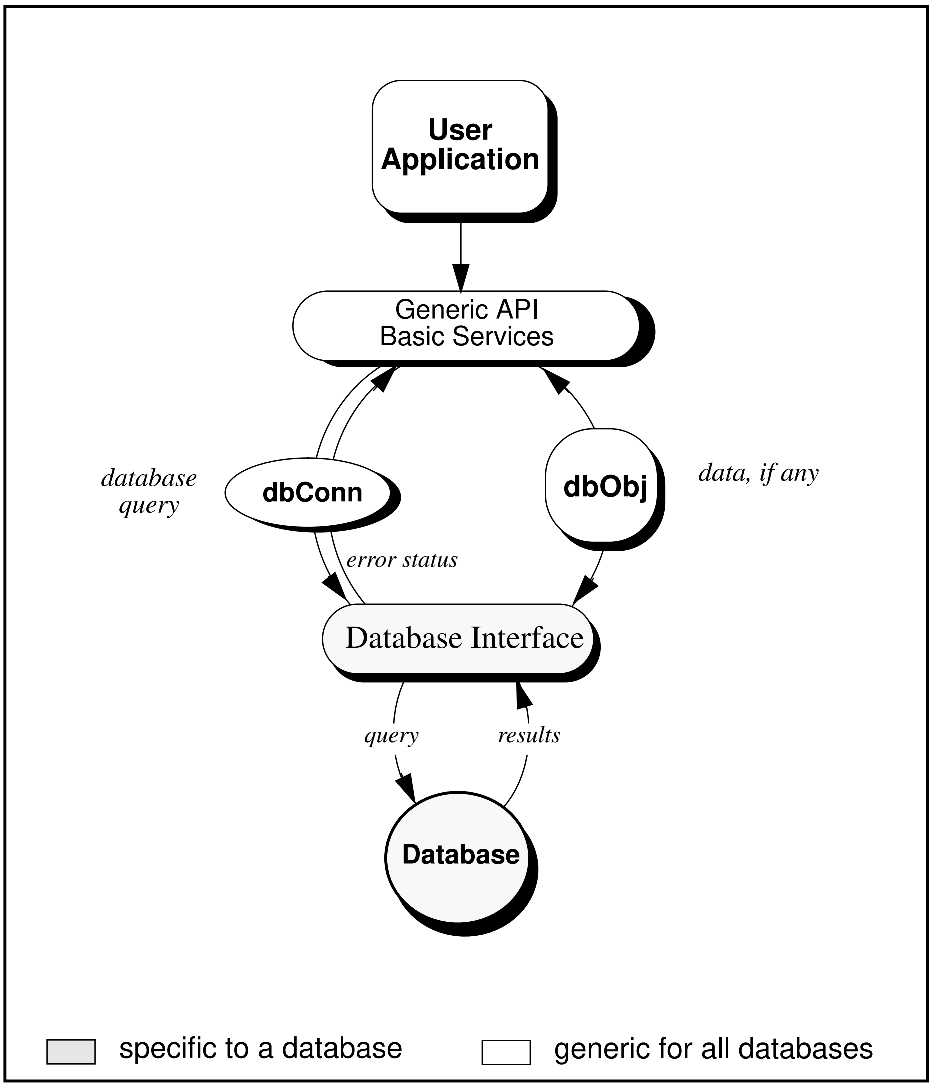

FIGURE 2. Generic Database Interface Architecture

2.1 Basic Services

GDI routines are organized into the following areas that provide:

1. *Communications*
   Database opens and closes, query cancellation, and query tracing.
2. *Error Handling*
   Consistent error reporting whether the actual error was a database error, a UNIX error, or an application-specific error. The application can decide whether warnings should be treated as fatal and a debug option automatically outputs errors to *stderr* to aid developers in debugging problems.
3. *Transaction Management*
   Hooks for starting a multi-statement transaction (POSTGRES and SYBASE), and for issuing commits, rollbacks, and savepoints.
4. *Data Dictionary Access*
   Consistent interface to each vendor's data dictionary for commonly asked questions such as "what is this object?", "what is its structure?", "who owns it?"
5. *Canned Database Queries*
   Highly optimized database access for commonly required functionality. For example, some vendor products have sequencing mechanisms while others do not. The *gdi_get_counter()* routine provides a highly optimized, consistent mechanism for fetching unique id's regardless of database.
6. *Dynamic Queries*
   Support for dynamic queries.
7. *Data Management*
   Data are managed in native application data format.

2.2 Database Connector (dbConn)

The Database Connector (dbConn) manages queries. When an application connects to the database, the GDI creates a dbConn that keeps track of administrative information, such as:

- database vendor type (*i.e.,* ORACLE, POSTGRES, SYBASE)
- database name, account, and node
- error information for the last query executed (specific database error code and string)

A single application can have multiple dbConn's, consisting of multiple connections to the same database or to a mixture of databases, as depicted in Figure 3.[1]

1. Only one connection to POSTGRES is allowed at this time, but an application may mix one POSTGRES connection with many ORACLE and SYBASE connections.

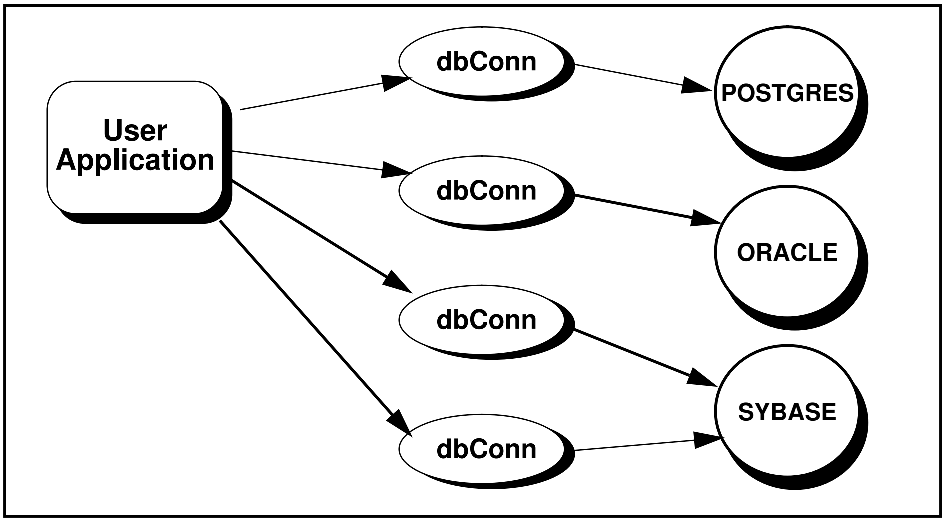

FIGURE 3. Database Connector (dbConn)

The dbConn also keeps track of the query *channel*, a communications “pipe” on which database queries are managed and executed. A channel is a DBPROCESS for SYBASE, a cursor for ORACLE, and a portal for POSTGRES. Each dbConn is initialized with at least one channel for default query activity, but users may add as many channels as they like, as depicted in Figure 4.

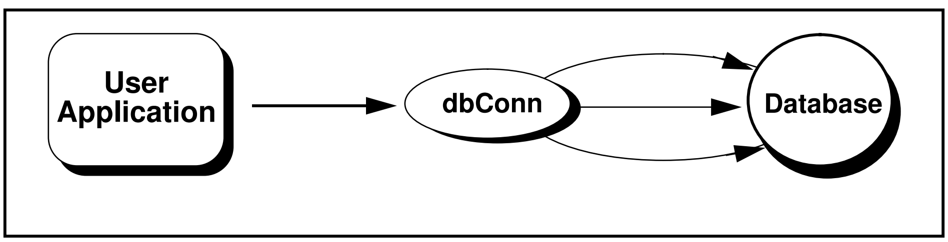

FIGURE 4. Database Query Channels

2.3 Database Object (dbObj)

The Database Object (dbObj), depicted in Figure 5, manages data and is composed of the following internal structures:

- *Tuple Container*
  Stores the data, which might be query results from a SELECT (outputs), or data to be inserted into the database (inputs). By default, data are organized into rows and columns, like a database table. The exact organization is controlled by the Tuple Constructor.
- *Column Definitions*
  Describes each column in the tuple container, including name, data type, and length.

- *Tuple Constructor*
  Specifies how to manage data in the tuple container. For example, S-PLUS operates on columns and rows instead of on rows and columns. The S-PLUS custom interface, described in Section 8, uses an S-PLUS tuple constructor instead of the default tuple constructor. While the specific data format is intended to be transparent to the end-user, Section 5.3 describes how programmers may create tuple constructors to fit a particular application need.
- *Query Information*
  Retains query information, such as the database query string, whether or not the query succeeded, and how many rows were affected. The dbObj retains general GDI information with each result set, while the dbConn stores specific database error information about the last query executed.

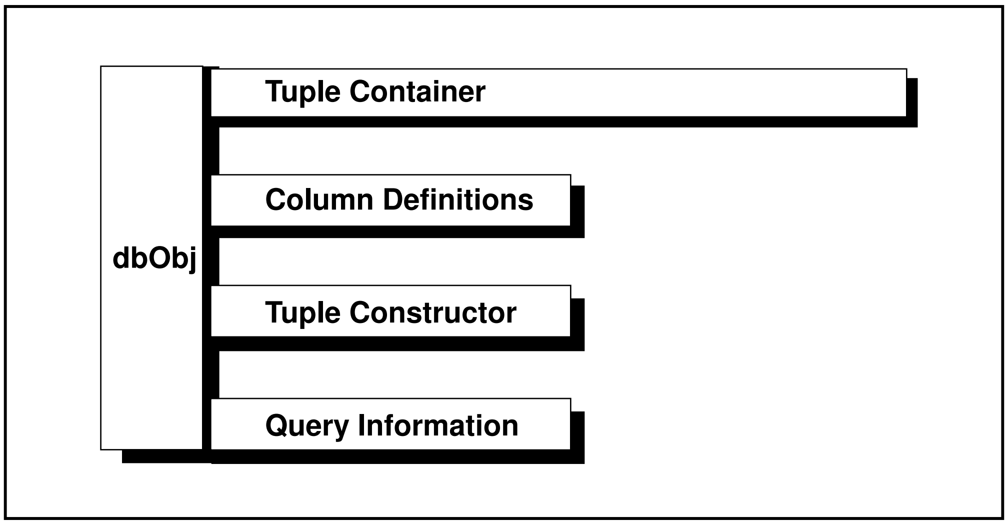

FIGURE 5. Database Object (dbObj)

The GDI provides functions and macros for accessing a dbObj. The user does not need to know the internal structure.

2.4 Comparison to Previous Interfaces

SAIC has developed several database library interfaces. They supported the most basic database services, the first five items discussed in section 2.1. But none of them supported fully dynamic queries and data management, resulting in two fundamental flaws:

- Libraries were Schema-Driven.
- Data structures were inflexible.

This section describes how the dbObj solved both these problems.

2.4.1 Schema-Driven Libraries

Fully dynamic database selects were difficult to support because there was not a straight-forward way to pass dynamic query results back to the calling application. Instead, insert and fetch routines, with the corresponding C and FORTRAN program headers, were generated automatically for each table based on its definition in the database. If the structure of the database changed, the push of a button would regenerate the support library.

In essence, the database access library was hard-coded to the schema being accessed, an approach that had serious limitations:

- *Poor Support for New or Changing Database Structures*
  Applications could not access newly created tables until headers and routines had been generated, the library remade and reinstalled, and the application recompiled. Modifying existing tables required synchronizing changes to database tables, access libraries, program headers, and the applications. The library became a weak link between the application and the database.

- *Inflexible SELECT Lists*
  Since the SELECT list was hard-coded to a single table, an application received all fields in a table even if it wanted just one. More importantly, an application queried one table at a time, even though it might need data from many tables. The application had to select from each table separately, then merge the results. Because of this, the number of application-specific routines grew, defeating one of the primary purposes of a centralized library which is to reuse code.

The dbObj overcomes the problem of managing dynamically defined query results. Applications may access new tables as soon they are created, access existing tables as they are changed, and execute any database statement that is legal for the target database.

2.4.2 Inflexible Data Structures

Previous interfaces supported one data structure: an array of structures. If an application needed a linked list, it constructed the list and copied the data into it. Likewise, data were copied to FORTRAN storage. Loading data into S-PLUS required dumping results to a flat file, then manually describing and loading the file into S-PLUS. Too many steps were required to migrate or copy data into the application.

The dbObj reduces data copying by supporting the application structure directly.

2.5 Restrictions

While an application may attach to multiple databases simultaneously, no effort is made to translate queries for the target database; the GDI passes the query straight through.

SQL Support

Commercial relational databases extend the ANSI SQL standard with features that are not guaranteed to work with other products. For example, a query containing the ORACLE outer join operator (+) will fail if it is sent to a SYBASE database which uses the asterisks (*) as the outer join operator.

The GDI passes database queries directly to the database. It does not parse nor translate queries to another vendor's SQL dialect. Vendor-specific features should be avoided. Appendix A notes which references describe ANSI SQL.

Transaction Management

Transaction management and query channels are handled differently by the various database vendors. Some functions are only applicable to a subset of the supported databases. Other functions have different affects depending on the target database.