# GEN: A Database Interface Generator for HPC Programs

Quan Pham, Tanu Malik
Computation Institute
University of Chicago and Argonne National Laboratory
Chicago, IL 60637
*quanpt@gmail.com, tanum@uchicago.edu*

## ABSTRACT

In this paper, we present GEN an interface generator that takes user-supplied C declarations and provides the necessary interface needed to load and access data from common scientific array databases such as SciDB and Rasdaman. GEN can be used for storing the output of parallel computations directly into the database and automates the previously used inefficient ingestion process which requires development of special database schemas for each computation. Further, GEN requires no modifications to existing C code and can build a working interface in minutes. We show how GEN can be used for Cosmology analysis programs to output data sets in real-time to a database and use for subsequent analysis. We show that GEN introduces modest overhead in program execution but is more efficient than writing to files and then loading. More significantly, it significantly reduces the programmatic overhead of learning new database languages.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Databases**]: Metrics—*complexity measures, performance measures*

## General Terms

Parallel programs, HPC systems, database systems

## Keywords

scientific data management, data-intensive computing, data organization

## 1. INTRODUCTION

Today, massively parallel computers supply enormous computational power for high-performance scientific applications that use them to run large-scale simulations. For most sciences, simulating a scientific phenomena is not enough in itself. To make scientific discoveries, the output from simulations must still be analyzed.

Simulations, for efficiency, output file datasets in write-optimized formats. To analyze the datasets, they must often be reorganized, such as by grouping, sorting, and joining attributes in a different order than they appear in the original datasets. Such reorganization of files and their attributes is necessary for efficient analysis but also introduces delays in executing the analysis step. In addition, the reorganization itself is often programmed in ad hoc ways either outside of or within the analysis program.

To reduce reorganization overhead post-simulation and prior to analysis, data can be ingested into a database, relational or otherwise, which provide standard abstractions for grouping, sorting, joining at improved efficiency. However, the use of a database within this workflow is currently not a standard practice. While there are many adoption bottlenecks, a primary reason is that databases must still be "plugged in" within the natural workflow of the file-based simulation and analysis programs. Thus even though they provide benefits for data reorganization the advantages are overlooked due to the initial overhead of "plugging in" the databases. While, of recently more scientific-friendly databases for storage of scientific data are being developed [2, 3], these databases continue to provide interfaces that requires a scientist to have an intimate knowledge of how the database works.

This "plugging in" can be challenging; the scientist must transition from the imperative language environment such as C/C++ to a declarative language environment used by databases. Even if they do the programmatic transition, they must still develop appropriate interfaces that convert the data model embedded in their programs to the data model of the database and subsequently ingest data from the self-describing, write-optimized files to analysis-optimized database. Oftentimes, this interface needs to be written for the myriad analysis that the scientist is doing; using databases increases more data management overhead than simplifying the process.

One possible solution for "plugging-in" is to simplify database interfaces and adopt a database-as-a-service approach. The service approach reduces typical configuration, installation, compilation issues associated with the database, It further automates data uploads by inferring schemas in files, and simplifying querying capability by exposing it through a RESTful API. However, servicing the database through RESTful APIS can introduce a significant performance overhead for a HPC workflow of simulation and analysis applications that typically leverage optimization in the operating system to read and write data. These analysis applications

*SSDBM'15* July 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2791347.2791363

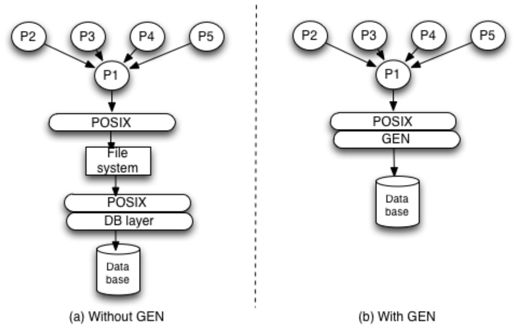

**Figure 1: Data insertion from HPC program to database without (left) and with GEN (right)**

would benefit more by interfacing the parallel programs directly with the database through the operating system layer rather than being redirected through an HTTP layer.

Towards this goal, in this paper, we present `GEN` an interface generator that takes user-supplied C declarations that are used at program run-time to load and access data from common databases. When a program is compiled with `GEN`, the appropriate DDL statements are generated and when the program is executed, the output of computations are stored directly into the database.

More specifically, without `GEN` data must be converted into a database-specific format and then ingested through some database loader. The scientists would need to create database DDLs, map data types, and interpret data to ingest them in appropriate tables for the database. This is an essential part of the organization process. (Figure 1(a)). Using GEN, database statements for a given program are generated automatically and corresponding file data is streamed directly to the database (Figure 1(b)). Currently the `GEN` prototype works for programs written in C and data intended for Postgres relational database and SciDB array database.

## 2. GEN OVERVIEW

We provide an overview of `GEN` through a motivating example and highlight its prominent features.

We consider an HPC program 'P' that computes the finite difference solution of 1D heat equation [1], and outputs an array of linearly spaced temperatures in the 1-D space at different times 2. The user would like to store the array output [1] in the database so it is easy to sub-select regions of the array in the spatial dimension and to be able to transform between the array from the time domain to the spatial domain and vice versa.

'P' is written in C language, but, in general, the program may be written in other compiled languages such as C++ or Fortran. Our focus is on the I/O part of the program i.e. the part of the program that writes the data output to files. We make two simplifying assumptions with respect to program I/O. First that the program performs I/O serially using the POSIX standard. Given that the program may run in parallel, the program may also choose to perform the I/O in parallel (using MPI-IO). However, in this paper, we restrict to serial I/O and keep parallel I/O for future work. Second, we assume that all I/O is done contiguously within the context of one C function definition, i.e. there are no straddled I/OS across multiple C function definitions or multiple `fseeks` that seek different locations of the file to write. Our experience with HPC programs shows that the above are safe assumptions for serial HPC programs. Figure 2 shows on the I/O part of the 'P' program. This program uses standard C libraries to write the time-dependent spatial vector, named table. The entire I/O is performed by a single rank, i.e. by a single running process of the program.

To store the table array instead in a database, the I/O part of the program must issue database statements instead of calling the POSIX library for writing data. Thus instead of calling the `fopen()` call to *x_data.txt*, we should be able to issue a create database object call to create a database object with the same name. Similarly, instead of `fwrite()`, we should be able to issue a corresponding database *insert* statement. While semantically this makes intuitive sense, there are two challenges:

- Since the C compiler cannot interpret database calls, the first challenge is to determine how to redirect the system calls to, instead, issue database statements.
- Owing to the structure maintained by databases, syntactically there is no direct mapping between POSIX function calls and the corresponding database statements (Figure 3). This raises the second challenge of determining the corresponding database schema.

In GEN, we solve the first challenge by using operating system directives, and the second challenge by utilizing runtime information and creating the correct database table in stages. We describe our implementation in Section 3, but in solving these challenges, we maintain two primary objectives:

**Introduce no changes in source code** Scientists, ideally, would not like to use a solution that reduces data management but is provided at the expense of introducing more programmatic overhead. Therefore, in GEN we wrap system calls and use the wrapper functions to create temporary tables that ingest some data. This data is later reorganized based on a target schema which is obtained from run-time write system calls and by using scientific file format libraries.

**Reduce operating system-database system impedance mismatch** Even if GEN does the right programming language syntax translation, writing data to a file in not the same as writing data to a database. In the operating system, write system calls are optimized to do I/O at the granularity of pages, where as in a database I/O is conducted at the granularity of records (as in relational databases) or cells (as in array databases). This impedance between the two systems makes it difficult to simply execute database statement instead of programming language statements. Data objects in operating system buffers must be interpreted and then inserted in the database. Since this interpretation can be expensive for large amounts of data, we present a *lazy interpretation* mechanism of OS buffers within databases. Our key intuition is that it is cheaper to interpret and reorganize data within a database than perform schema translations outside it.

## 3. GEN IMPLEMENTATION

[1]Depending upon the analysis, different data structures may be output, such as trees, graphs, and spatial objects.

```
MPI_Init ( &argc, &argv );
MPI_Comm_rank ( MPI_COMM_WORLD, &id );
MPI_Comm_size ( MPI_COMM_WORLD, &p );

if ( rank == 0 )
{
    x_file = fopen ( "x_data.txt", "w" );
for ( j = 0; j < n; j++ )
   {
      for ( i = 0; i < m; i++ )
        fprintf ( output, "  %24.16g", table[i+j*m] );

      fprintf ( output, "\n" );
   }
    fclose ( x_file );
}
```

**Figure 2: A sample program I/O**

```
MPI_Init ( &argc, &argv );
MPI_Comm_rank ( MPI_COMM_WORLD, &id );
MPI_Comm_size ( MPI_COMM_WORLD, &p );

if ( rank == 0 )
{
    CREATE TABLE "x_data.txt"
    for ( j = 0; j < n; j++ )
    {
      for ( i = 0; i < m; i++ )
        INSERT into "x_data.txt" VALUES (table[i+j*m])
      fprintf ( output, "\n" );
    }
    fclose ( x_file );
}
```

**Figure 3: Incorrect Database Interface of Sample Program**

To generate the correct interface, GEN relies on (a) run-time wrapping of POSIX I/O calls, (b) run-time bookkeeping information for creation of database DDL statements, and (c) ingesting data directly from OS buffers and lazily interpreting them within the database using scientific file-format libraries. We explain this interface generation in detail:

## 3.1 Runtime Wrapping of POSIX I/O calls

Since we do not want to change the user's program, we would like that whenever the C program makes a POSIX system call's, the call be directed to GEN's library to make a corresponding database system call. In C programs, I/O system calls are handled through the system libraries, such as stdio or iostream which are shared libraries. A shared library may be loaded during execution into the memory space of the program's process or may already be loaded into the memory space of another process. Further, a program process can be forced to load a shared library using the `LD_PRELOAD` environment variable, which instructs the loader (ld.so) to load the specified shared libraries. Symbol definitions in `LD_PRELOAD`ed libraries will be found before definitions in non-`LD_PRELOAD`ed libraries, so if an `LD_PRELOAD`ed library contains a definition of, say, `fopen`, then that code will be executed when the process references `fopen`. In particular, this means that we can package our wrapper functions into a shared library (`.so` file) and use `LD_PRELOAD` to cause them to be loaded.

The `LD_PRELOAD` directive helps to direct system calls to GEN's wrapper functions, but at run-time GEN's wrapper functions still does not have the handle to the original system call functions, the arguments of which are necessary to create corresponding database DDL statements in the wrapper function and then execute them.

## 3.2 Creating Database DDL statements

To create database DDL statements we need the handle to the original system call function and use the parameters of the original system call function create appropriate database DDL statements. On Unix platforms, the `dlsym` function `dlsym(void* handle, char* symbol)` returns the address of the symbol given as its second argument, searching the dynamic or shared library accessible through the handle given as its first argument. Thus, assuming the only definitions of fopen are in our shared library and in the system C library, calling `dlsym(RTLD_NEXT, "fopen■)` will return the address of the standard `fopen()` function, to which we may assign a function pointer and later call. In this `RTLD_NEXT` allows searching for libraries in the entire library path.

Given the handle to system call functions, GEN creates appropriate database DDL statement, based on the so far available information. Thus when the fopen system call is available at run time,

```
fopen("x_data.txt", w")
```

GEN redirects it to the following DDL statement

```
Create <Database Object> x_data.txt.tmp <structure>
```

where <database object> refers to keywords TABLE or an ARRAY. Typically, in a database the above declaration is incomplete since it does not have the associated structure of the database object. In case of a TABLE that structure corresponds to the attributes and their types of the TABLE, and in case of ARRAY the structure corresponds to the dimensions and attributes of the ARRAY. Since the structure information is not known at the time of opening a file, we add dummy attributes and dimensions to make it complete. Thus for relational databases the structure adds an integer column, which is auto-incremented, and a column of binary data type. For scientific databases the structure adds a uni-dimensional array with a single attribute of type integer. While, creation of these tables may seem unnecessary at this point, in next subsection we describe how these tables can be used for lazy reorganization of the data within the database.

When the *fwrite()* system call is redirected to GEN, then GEN uses the handle of *fwrite* obtained from `dlsym` to obtain the actual data structures that will be written out in the file. GEN obtains the declarations of these data structures from the header files to initialize the appropriate schema for the database objects that it created earlier. At this point, instead of updating the schema of the previous tables, it creates companion table/array with the correct data structure.

Thus given a fwrite declaration, such as

```
fwrite(&data, sizeof(example), 1, fout);
```

`GEN` determines the data structure to which *data* belongs to, i.e., *example* and using its declaration specified in header files obtains the appropriate attributes and type information. Thus, all data structures are flattened out as attributes of a table in a relational database, and as attributes of a single-dimensional array in a array database. Thus nested data structures are two tables joined by a foreign key. Using the same logic, but some more bookkeeping we can create appropriate statements for *fprintf* statements and *sprintf* statements.

### 3.3 Ingesting Data Directly From Program to Database

`GEN` still does not create the corresponding INSERT/LOAD statements for the table due to the operating system-database system impedance mismatch where in a *fwrite()* in a C program does not correspond to an INSERT/LOAD statement in a database. This is due to use of system libraries in which writes are buffered before sending to the OS kernel and the OS kernel converts these into streaming writes to improve performance.

IN `GEN` we continue to maintain this performance advantage and create an INSERT statement for the *fwrite*, but direct the data for the *fwrite* into the temporary table of the file to which *fwrite* intends to write. Thus for the previous *fwrite* statement the following INSERT statement is created:

```
INSERT INTO x_data.txt.tmp \
VALUES(id, <data pointed by &data>)
```

While the writes are being sent to the temporary table, GEN maintains two pieces of information: it maintains what data structure is being written out and it maintains the number of times, *fwrite* was redirected by `dlsym`. The latter information is important for us to lazily reinterpret the data structure using the scientific file format libraries. For array databases, the process of ingesting data is not similar. Data from *fwrite* must be interpreted before it can be inserted. Thus array databases have significantly more delay in inserting data than relational databases.

## 4. EXPERIMENTS

In `GEN` we incur two kinds of costs for interface generation: the cost of redirecting the POSIX system calls during the run-time, and the cost of inferring the data structure lazily. We measure these costs individually. We also measure the cost of not using `GEN` and loading data via files.

We have build a working prototype in which MPI programs, similar to the one in the example, stream data to a Postgres database and a SciDB database. Both Postgres and SciDB are installed in a single-instance mode on a single node. The MPI program is run in parallel on 3 nodes with one node streaming data to the database using GEN. Experiments were conducted on Ubuntu machines with 4 GB RAM and dual-core processor.

To measure the overhead of redirection, we experimented by increasing the number of dynamically loaded libraries associated with a program. Our experiments show that redirection increases the program run time by 0.04ms with a variation of 0.02. The redirection time is not influenced by increasing the number of elements to write in the program, since the libraries are already loaded in memory. The variation comes if we execute other programs, which can offload the library from memory and it needs to reloaded again.

In the second experiment, we measured the space vs time overhead of reorganizing the data. In `GEN` we have not saved any space. Without `GEN` the binary data is written out on files and then within the database. With `GEN` the data is written out to temporary tables within the database and then reorganized within the database. However, as Table 1 shows that there is significant savings in terms of reorganizing the data within the database versus reorganizing prior to loading the database.

Table shows the time it takes to load increasingly larger sized datasets into a relational and an array database vs how much time it takes to load the same data from a temporary table into a formatted table plus the time of loading the initial binary data into a temporary table.

**Table 1: Reorganization times**

| Data Size | Postgres | | SciDB | |
|---|---|---|---|---|
| (in GB) | Load from Files | Reorganize within DB | Load from Files | Reorganize within DB |
| 1 | 2.7 | 1.2 | 8.4 | 3.2 |
| 5 | 6.8 | 2.5 | 14.3 | 7.6 |
| 10 | 10.56 | 4.7 | 22.6 | 8.9 |

## 5. RELATED WORK

Typically, simulation data is written onto data files in write-optimized, self-describing formats to be analyzed later. However, the increasing volumes of data and the need to reduce the time to analysis, different analysis approaches for analysis have emerged. In [4] and [5] data is not reformatted for analysis, but enabled over write-optimized data files. In [4], this is achieved by integrating the self-describing scientific file formats with the HDFS file system, and subsequently using map-reduce methods for analysis. In [5], analysis is enabled by translating common queries, written in a native language, into appropriate library calls of the scientific file formats. In both cases, only simple sub-selections, and aggregations are enabled. Databases allow more complex analysis. However ingesting data into a database and monitoring it can be onerous. In this paper, we have reduced this overhead by directly linking HPC programs with database statements, and thus automating the process for simulation based science that generates a lot of data but would like to automatically ingest data into databases instead of files.

The operating system provides a different and simplified model for writing data. Optimizations in the operating system have long been discussion of the systems research community. In this paper, we have kept most of the optimization that the OS system provides, and yet provided a database that can slowly be adapted on a need basis for analysis. While our approach is preliminary, we plan to extend it to different operating systems such as Windows and Mac OS X and determining appropriate system libraries.

## 6. CONCLUSIONS

In this paper, we described a method to transparently introduce and link databases within the natural workflow of

HPC programs The method has two fundamental characteristics (i) to enable scientists to use the database without changing their source codes that typically use POSIX-style I/O, (ii ) to lazily reorganize file datasets and extract meta information from a more database-style schema. The primary objective of conceiving this method was to encourage use of databases amongst HPC scientists without introducing a database language learning curve.

## 7. REFERENCES

[1] Burkardt, J. “Parallel Programs” `http://people.sc.fsu.edu/~jburkardt/c_src/fd1d_heat_explicit/fd1d_heat_explicit_prb.c`

[2] Brown, Paul G. “Overview of SciDB: large scale array storage, processing and analysis”, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010.

[3] Baumann, Peter, et al. “The multidimensional database system RasDaMan”, ACM SIGMOD Record, Vol. 27, No. 2, ACM, 1998.

[4] Buck, Joe B., et al. “SciHadoop: Array-based query processing in Hadoop” Proceedings of 2011 International Supercomputing Conference, ACM, 2011.

[5] Su. Y., and Agrawal, G. “Supporting User-Defined Subsetting and Aggregation over Parallel NetCDF Datasets”, In Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, 2012.

[6] Howe, Bill, et al. “Database-as-a-service for long-tail science”, In Scientific and Statistical Database Management, (SSDBM), 2011.

[7] Tanenbaum, Andrew S., and Albert S. Woodhull. Operating Systems: Design and Implementation. Vol. 2. Englewood Cliffs, NJ, Prentice-Hall, 1987.

## APPENDIX

## A. RUN-TIME FUNCTION WRAPPING

Here, we demonstrate the `LD_PRELOAD` run-time function replacement technique by providing a wrapper for the `fopen` function. This example was tested on RedHat version 9.

The full definition of the `fopen` function is `FILE *fopen(const char *path, const char *mode);` it opens a new file stream on the given path.

First, we need to write the wrapper function, which we place in `fopenwrap.c`. Note that under Linux, we need to `#define GNU SOURCE` to have access to the `RTLD_ NEXT` pseudo-handle.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
FILE* fopen(const char* path, const char* mode) {
        printf("Opening %s\n", path);
FILE* (*real_fopen)(const char*, const char*) =
        dlsym(RTLD_NEXT, "fopen");
return real_fopen(path, mode);
}
```

Next, we compile the wrapper into a shared library.

```
gcc -fPIC -rdynamic -c fopenwrap.c
gcc -shared -o libfopenwrap.so fopenwrap.o -lc -ldl
```

Now, we create the file fopen.c, containing test code for our wrapper.

```
#include <stdio.h>
int main(int argc, char** argv) {
    char buf[80];
    FILE* fileptr = fopen(argv[1], "r");
    do {
        fgets(buf, 80, fileptr);
        if (feof(fileptr)) {
break; }
        printf("%s", buf);
    } while(1);
    fclose(fileptr);
return 0; }
```

Finally, we compile the test code and run it with our wrapper library.

```
LD_LIBRARY_PATH
```

tells the loader to look for shared libraries in the current directory.

```
gcc fopen.c -o fopentest
LD_LIBRARY_PATH=. LD_PRELOAD=libfopenwrap.so \
./fopentest fopen.c
```

This will print “Opening fopen.c” followed by the source of the test program.

## B. LINK-TIME FUNCTION WRAPPING

We again provide a wrapping function for fopen, except this time we use the link-time wrapping method. This example was tested on RedHat version 9.

First, we need a C file (`fopenwrap.c`) containing the wrapper for `fopen`:

```
#include <stdio.h>
FILE* __wrap_fopen(const char* path, const char* mode) {
        printf("Opening %s\n", path);
return __real_fopen(path, mode);
}
```

We compile it to a static library:

```
gcc -c fopenwrap.c
ar rcs libfopenwrap.a fopenwrap.o
```

We will use the same test code as in the previous example, but we will compile it differently: Note how we use the `-Xlinker` option to `gcc` to pass arguments through to the linker.

```
gcc fopen.c -o fopentest -L. -lfopenwrap -Xlinker \
--wrap -Xlinker fopen
```

Running the test program with a readable file as its only argument will print the line “Opening <file>“ followed by the contents of the file.