ROOT I/O FOR SQL DATABASES

S. Linev, GSI, Darmstadt, Germany

Abstract

ROOT [1] framework of CERN has a powerful I/O system. It allows storing any objects in ROOT files and supports class version control via automatic schema evolution. Up to recent time only binary format was supported. At the same time in HEP community grows usage of SQL databases for storage of different kind of data like detector geometry, experiment conditions and so on, therefore a demand for SQL object store facility exists.

This paper describes the new TSQLFile class of ROOT, which provides a full-functional TFile interface to SQL database. Implementation details, table design and performance issues are discussed.

MOTIVATION

SQL (Structured Query Language) is the most popular computer language used to create, modify and retrieve data from relational database management systems (RDBMS).

ROOT provides abstract TSQLServer interface, which allows text queries execution and manipulation of retrieved result sets. For the moment, the following RDBMS are supported: MySQL [2], Oracle [3] and PostgreSQL [4]. With plain SQL queries one can manipulate tables with basic data types, but what can be done to store complex data (like ROOT objects) in relational database?

First of all, one can implement custom I/O code for user classes. Depending on complexity of user classes, this approach requires intensive development and support (maintenance) of I/O code. Of course, potentially such solution can be the best in means of performance.

Another possibility for object I/O is the usage of existing software frameworks [5]. Such framework eliminates implementation problems, but it typically implies strict coding, data type usage and class design rules, which may be incompatible with ROOT I/O scheme.

ROOT I/O already has very powerful and well established object I/O system therefore one can consider to use native ROOT I/O for SQL database. ROOT provides two concepts for data store organization: TTree and TFile classes.

TTree is suitable for storing of big amounts of similar data like events. Tree structure with branches and leaves one can very well map to the structure of one (or several) database tables. Such approach was implemented in ROOT in TTreeSQL class. It allows updating of table content and supports normal tree-draw mechanism. In current implementation only basic data types are supported, therefore one cannot automatically map a class structure to TTreeSQL and use it as object store.

TFile class provides hierarchical storage for arbitrary objects. Subdirectory structure and keys mechanism allow partial access to data in the file. Automatic schema evolution supports different versions of the same class, stored in the file at different times. Implemented for SQL backend, one can benefit both from standard ROOT I/O interface and from reliability and accessibility of modern databases. Common TFile interface for different backends (binary file, XML file, SQL database) will allows easy developing and testing of user I/O code even with very complex objects like detector geometries.

IMPLEMENTATION

Table design

Before one can stream an object into SQL database, one should define structure of the tables where the data will be stored. There are many ways to map class hierarchy and complex data types into database tables [5]. For this development it was important that tables structure corresponds to existing ROOT I/O strategy and does not impose any limitations. At the same time tables structure should be clear and data should be easy accessible.

A strategy with one table per class version and one column per elementary data member was taken. ROOT I/O is organized in such a way, that each class has a streamer which stores class members in a buffer. Therefore for SQL I/O case class data will be streamed in context of one table. Differences in class versions typically indicate changes in class structure, therefore different tables should be used for different class versions. The table name can be composed like *<ClassName>_<ClassVersion>*. For example, for current version of TObject class, the table name is *TObject_ver1*. Other examples: *TH1_ver5*, *TAxis_ver9*.

Each row in the class table corresponds to one object instance. To identify objects, the first column of the class table contains an unique object identifier. Other columns contain class member data and depend from member type.

A basic data type will be mapped to an appropriate SQL type and stored in a single column. For instance, *int* type can be stored in column of SQL type INT, float in column of type FLOAT, and so on. Name of such column should be derived from data member name.

An array of basic types of size N can be mapped to N columns in the table. The only limitation is that N should be a constant. Column names will contain the array index.

With the chosen strategy of one table per class, data of parent classes should be stored in corresponding tables, using same unique object id. To unambiguously identify

where data of the parent class is stored, the class table should include for each parent class a column with current version of parent class. For example, *TH1I_ver1* table should contain column of name *TH1*.

More advanced treatment should be done for class members like object instances or object pointers. One cannot (and would not) store complete object in a single table column. Instead, one can store such component with another identifier in appropriate tables and only that identifier should be kept in the appropriate column of class table. Similar technique can be done for fixed array of objects and objects pointers.

But not all class members can be represented in "normalized" form in SQL tables. Arrays with variable size or custom streamer data do not fit into one column per class member strategy and should be represented differently. For such data special tables with names like *TGraph_streamer_ver4* should be created. Data in such table is stored row-wise, i.e. each elementary value like an array entry, will use one row in that table. This data can be stored in text format only. In more details, the structure of such tables is described in section "examples".

TBufferSQL2 class

The standard streamer mechanism of ROOT uses a TBuffer object to convert class member data to/from binary format. To access object data and produce correspondent SQL queries, the new TBufferSQL2 class was introduced. It redefines most TBuffer virtual methods.

When writing object data, TBufferSQL2 converts streamed values (integers, floats and so on) to text and composes SQL INSERT queries for appropriate tables. Name and type of columns are derived from class member info, provided to TBufferSQL2 by ROOT streaming functions.

When reading object data, TBufferSQL2 produces appropriate SELECT queries to retrieve object data from SQL tables first and then performs conversion from the text to object data members.

Custom streamer support

A custom streamer necessary when standard ROOT I/O cannot correctly store class data or the automatic schema evolution is not able to process all previous class versions. In that case exact I/O code for class data members should be written by user and will differ from standard I/O calls. There are a lot of classes in ROOT itself with custom streamers, therefore it is important to support them for SQL I/O.

The main difference from standard ROOT I/O is that custom streamer does not provide meta-information (member names and types) for streamed data, which would allow splitting data into columns. Therefore data, produced by custom streamer, is converted value-by-value into text format and is streamed into a special table. This table contains two columns: *Filed* – indicates value type (integer, version, char*) and *Value*. Such table created for each class version (when required). For instance, data of class TList will be stored directly in table *TList_streamer_ver5*.

TSQLFile class

This is the core class of SQL I/O in ROOT. The main goal of that class is to provide a complete TFile interface to SQL databases.

For connection to database a virtual interface was used, provided by TSQLServer, TSQLResult and TSQLRow classes. It allows text query execution on any supported SQL database and provides uniform access to produce result sets.

When user writes an object to TSQLFile, a special TKeySQL object is created (similar to TKey in TFile) and its data is stored in *KeysTable*. This table contains all keys of all file subdirectories.

Each streamed object is also registered in *ObjectsTable*, where object id, class name and version are indicated. From this data one can define in which table data of streamed object is contained. *ObjectsTable* also contains key identifier, to which object belong to.

In addition to standard TFile functionality, TSQLFile provides database-specific features. For instance, user can specify usage of database indexes for class tables. By default, indexes are created only for keys and objects tables. As option, column names in class tables can have suffixes indicating type of column (parent class, object, integer and so on). Different modes of transaction usage can be specified.

User also can configure usage of SQL transactions. By default, writing of complete key data is supplied by START TRANSACTION – COMMIT commands. User can switch this mode off (if database does not support transactions) or apply commands himself for bigger blocks of write operation.

For MySQL database the type of the tables can be indicated. Any of these configurations can be changed only before first object is written do TSQLFile. All configurations are stored in small *Configurations* table.

Automatic schema evolution

Automatic schema evolution of ROOT allows to read class data even if the structure of class members has been changed. To be able to read data that was written by old class versions, ROOT keeps a list of streamer info objects, where description of each persistent class member is contained.

As a TFile class, TSQLFile preserves all used streamer infos and stores them in the database as special key object, not seen by user. Automatic schema evolution functions were adjusted to support also text-based streaming (TSQLFile and TXMLFile backends).

EXAMPLES

Example with TBox class

Let's consider a small macro, which creates TSQLFile instance and store several TBox objects in the tables.

```
{
   TSQLFile f("mysql://host.domain/test",
                  "create","user","pass");
   for (int n=1;n<=10;n++) {
      TBox* b = new TBox(n,n*2,n*3,n*4);
      b->Write(Form("box%d",n));
   }
}
```

TBox class inherits from TObject, TAttLine and TAttFill classes, and declares four Double_t members. Figure 1 shows content of *TBox_ver2* and *TObject_ver1* tables, produced by the macro.

TBox_ver2

obj:id	TObject	TAttLine	TAttFill	fX1	fY1	fX2	fY2
1	1	1	1	1	2	3	4
2	1	1	1	2	4	6	8
3	1	1	1	3	6	9	12
4	1	1	1	4	8	12	16
5	1	1	1	5	10	15	20
6	1	1	1	6	12	18	24
7	1	1	1	7	14	21	28
8	1	1	1	8	16	24	32
9	1	1	1	9	18	27	36
10	1	1	1	10	20	30	40

TObject_ver1

obj:id	UniqueId	Bits	ProcessId
1	0	50331648	
2	0	50331648	
3	0	50331648	
4	0	50331648	
5	0	50331648	
6	0	50331648	
7	0	50331648	
8	0	50331648	
9	0	50331648	
10	0	50331648	

Figure1. *TBox_ver2* (left) and *TObject_ver1* tables

In both tables one can see *"obj:id"* columns, containing unique object identifier. Columns *fX1, fY1, fX2, fY2* in table *TBox_ver2* contains value of class data members while columns *TObject, TAttLine, TAttFill* contains versions of parent classes. TObject class streamer has special treatment in SQL I/O, therefore column names differ from TObject class member names.

At the same time *KeysTable* is created, where all instances of stored object are registered (see Figure 2).

KeysTable

key:id	dir:id	obj:id	Name	Title	Datime	Cycle	Class
10	0	1	box1		2006-02-01 14:52:36	1	TBox
11	0	2	box2		2006-02-01 14:52:36	1	TBox
12	0	3	box3		2006-02-01 14:52:36	1	TBox
13	0	4	box4		2006-02-01 14:52:36	1	TBox
14	0	5	box5		2006-02-01 14:52:36	1	TBox
15	0	6	box6		2006-02-01 14:52:36	1	TBox
16	0	7	box7		2006-02-01 14:52:36	1	TBox
17	0	8	box8		2006-02-01 14:52:36	1	TBox
18	0	9	box9		2006-02-01 14:52:36	1	TBox
19	0	10	box10		2006-02-01 14:52:36	1	TBox

Figure2. *KeysList* table content

Each key entry contains identifier, id of parent directory, object id, key name and title, creation time, cycle number and object class name.

User access to produced tables

The data, as produced with previous example, can be read with standard ROOT I/O code:

```
{
   TSQLFile f("mysql://host.domain/test",
                  "open","user","pass");
   for (int n=1;n<=10;n++) {
      TString name = Form("box%d",n);
      TBox* b = (TBox*) f.Get(name);
   }
}
```

But the user can also read data from tables with any other SQL browsing tools. TSQLFile provide MakeSelectQuery() method, which allows to combine object data from different tables in one SELECT statement. For example, result of such statement for TBox class can be seen on Figure 3. One can see data of TObject, TAttLine, TAttFill and TBox classes together.

TObject_ver1			*TAttLine_ver1*			*TAttFill_ver1*		*TBox_ver2*			
obj:id	UniqueId	Bits	fLineColor	fLineStyle	fLineWidth	fFillColor	fFillStyle	fX1	fY1	fX2	fY2
1	0	50331648	1	1	1	19	1001	1	2	3	4
2	0	50331648	1	1	1	19	1001	2	4	6	8
3	0	50331648	1	1	1	19	1001	3	6	9	12
4	0	50331648	1	1	1	19	1001	4	8	12	16
5	0	50331648	1	1	1	19	1001	5	10	15	20
6	0	50331648	1	1	1	19	1001	6	12	18	24
7	0	50331648	1	1	1	19	1001	7	14	21	28
8	0	50331648	1	1	1	19	1001	8	16	24	32
9	0	50331648	1	1	1	19	1001	9	18	27	36
10	0	50331648	1	1	1	19	1001	10	20	30	40

Figure3. Composition of TBox data

Example with TGraph class

Let's consider following macro, where TGraph object is created and stored:

```
{
   TSQLFile f("mysql://host.domain/test",
                  "update","user","pass");
   TGraph* gr = new TGraph(5);
   for (int n=0;n<5;n++)
      gr->SetPoint(n, n+1, (n+1)*(n+1));
   gr->Write("gr");
}
```

Several tables, produced by that macro, are presented in Figure 4. In table *TGraph_ver4* one can see columns corresponding to parent classes (*TNamed*, *TAttLine*, *TAttFill*, *TAttMarker*) and members of basic types like *fNpoints*, *fMinimum*, *fMaximum*.

TGraph_ver4

obj:id	TNamed	TAttLine	TAttFill	TAttMarker	fNpoints	fX	fY	fFunctions	fHistogram	fMinimum	fMaximum
1	1	1	1	1	5	0	6	2	0	-1111	-1111

TGraph_streamer_ver4

obj:id	raw:id	Field	Value
1	0	fX:Char_t	1
1	1	[0]:Double_t	1.000000
1	2	[1]:Double_t	2.000000
1	3	[2]:Double_t	3.000000
1	4	[3]:Double_t	4.000000
1	5	[4]:Double_t	5.000000
1	6	fY:Char_t	1
1	7	[0]:Double_t	1.000000
1	8	[1]:Double_t	4.000000
1	9	[2]:Double_t	9.000000
1	10	[3]:Double_t	16.000000
1	11	[4]:Double_t	25.000000

ObjectsTable

key:id	obj:id	Class	Version
10	1	TGraph	4
10	2	TList	5

TList_streamer_ver5

obj:id	raw:id	Field	Value
2	0	TList:Version	5
2	1	TObject:Version	1
2	2	UInt_t	0
2	3	UInt_t	50331648
2	4	UChar_t	0
2	5	Int_t	0

Figure4. TGraph class tables

But there are several TGraph class members, which cannot be represented in a simple way. For example, *fX* and *fY* is an array of double values *fNpoints* elements each. Such data is converted in "raw" format and streamed into *TGraph_streamer_ver4* table. In this table one can see data of two arrays, each 5 elements long. Each row in this column has identifier *"raw:id"*, which is used later for referencing in class table. Here *fX* and *fY* columns contain id of *TGraph_streamer_ver4* table row, where correspondent data is started. Row identifier

"raw:id" is also used to preserve sequence of values when data will be read from the table.

TGraph class member fFunctions contains pointer on TList class. This object is streamed separately with other id and registered in *ObjectsTable*. Because TList class has a custom streamer, its data is directly streamed into *TList_streamer_ver5* table and a normal clas table is not created. Column *fHistogram* contains pointer on TH1 class. Zero value here means that no histogram was assigned to it.

PERFORMACE

There are two different aspects in performance of SQL I/O. First, how many CPU time is used to convert object data to SQL queries and, second, how good/bad produced SQL queries and how long execution of that queries takes on server side.

A test was done with TClonesArray containing 10000 TBox objects. Such array was written to file and than read back. Same code was tested with TFile, TXMLFile and TSQLFile. CPU time, real time and number of SQL queries are show in table 1.

Table 1: Performance measurements

	Write			Read		
	CPU	Real	N	CPU	Real	N
Binary	0.03s	0.03s	-	0.02s	0.02s	
XML	1.21s	1.21s	-	1.44s	1.44s	
MySQL	3.23s	6.33s	60	2.54s	3.05s	6
Oracle	10.7s	186s	70019	7.97s	18.3s	6
Oracle*	2.86s	9.08s	14	4.31s	4.43s	6

Oracle* uses new TSQLStatement classes

For tests two RDBMS was used. First is MySQL 4.1, running on Fedora Core 4 Linux PC (Athlon XP, 1.15 GHz, 512 MB RAM). Second is Oracle 10g RAC, running on Linux SuSE 8 (Dual Xeon 2.4GHz, 4Gb RAM) were used.

It can be seen that binary ROOT I/O (as expected) much faster compared to text-based XML and SQL I/O. The main reason is that XML and SQL requires text conversion and intermediate buffering of all values. The size of produced data (XML structures or SQL queries) is also much bigger (~20 times) compare to compressed binary format.

At the same time, SQL I/O code was optimized to reduce as much as possible the number of queries supplied to SQL server. Where it was possible, access to the same table was done with single SQL query.

There are differences between MySQL and Oracle in the way how queries are submitted to the server. Current implementation of TSQLServer classes of ROOT allows only pure text queries. SQL syntax of MySQL allows multiple rows insertion in single query even in text mode, therefore with very long queries one can insert hundreds of rows in one communication loop to the server and gain performance. Oracle does not provide such possibility. Therefore one should apply hundreds of queries (one per row) instead. From table 1 one can see, that storage of 10000 objects will require about 70000 separate queries. As a result, performance of data writing of standard Oracle plugin in ROOT is very poor.

One definitely requires a more advanced interface which allows formulating more complicated queries. OCCI (Oracle C++ Call Interface), used in ROOT, already has such possibilities, but they are hidden via narrow interface of TSQLServer/TSQLResult classes.

New abstract TSQLStatement class was introduced to allow more advanced features of queries and results set handling. A test implementation for Oracle and MySQL was done. Results, shown in table 1 as Oracle*, were measured with usage of this new class. Performance in that case is much better and comparable with MySQL.

TSQLStatement class also can be extended for basic data types support like integers or floats. This will exclude multiple text conversions and also will improve performance. Currently TSQLStatement class is in development and not yet in ROOT distribution, but can be easily integrated into ROOT.

Besides performance question, there is a problem of support of others RDBMS in ROOT. Historically, TSQLServer classes were developed based on MySQL text-based C API, and later compatible support was implemented for Oracle and PostgreSQL. To extend number of supporting databases, one can provide implementation of TSQLServer classes based on ODBC – Open Data Base Connectivity interface [6]. It can be also alternative for native MySQL API, while ODBC provides much more extended features.

CONCLUSION

TSQLFile class provides new possibility for usage of SQL databases with ROOT. It simplifies developing and maintenance of user code and allows easy navigation and access of user data in SQL tables. For better performance and portability further development of database support classes should be done.

ACKNOWLEDGEMENTS

I would like to thank Philippe Canal and Rene Brun for their strong support of this work.

REFERENCES

[1] http://root.cern.ch/.
[2] http://www.mysql.org/.
[3] http://www.oracle.com/.
[4] http://www.postgresql.org/.
[5] http://en.wikipedia.org/wiki/Object-relational_mapping.
[6] http://en.wikipedia.org/wiki/ODBC.