

FairRoot Virtual Database

User Manual

November 26, 2013

Denis BERTINI
GSI - Scientific Computing
<http://fairroot.gsi.de>

Abstract:

This report describes the November 2013 version of the FairRoot Virtual Database library which allows the user to store and access conditions data using different database technologies. The first part introduces the basic concepts and the design of the database interface. The second part complements this initial information with installation issues and finally the third part describes the interface features in details and how to use them.

Contents

I	Basic concepts	1
1	The Virtual Database	2
1.1	Introduction	2
1.2	Problem statement and motivation	3
1.2.1	Files versus database	3
1.3	Database in practice	4
1.3.1	Work overheads	5
1.3.2	Selecting a database	5
1.4	New database technologies	7
1.4.1	The CAP theorem	7
1.4.2	Eventual-consistency	7
1.5	Design	9
1.5.1	Interface as compromise	9
1.5.2	Architecture	10
1.5.3	FairRoot integration	11
1.6	Parameter data types	12
1.7	Mapping data objects to tables	13
1.8	Version management	17
1.8.1	Temporal database	17
1.8.2	FairRoot initialisation scheme	18
1.8.3	Validity time interval	19
1.8.4	Validation basic rules	20
1.8.5	Relational model implementation	21
1.9	Persistency scheme	24
1.9.1	SQL processing	24
1.9.2	Disk cache	25
1.10	Connection Pooling	26
II	Database Guidelines	28
2	Handling a Database	29
2.1	Installing	29
2.1.1	Local @ GSI	29
2.1.2	Standalone system	29
2.1.3	MySQL	30
2.1.4	PostgreSQL	33
2.1.5	Oracle	34
2.2	Handling a database server	37
2.2.1	Managing accounts	39
2.2.2	Server Checks	40

III	User Manual	42
3	Basic Settings	43
3.1	Setting up the Environment	43
3.2	The SQL-IO Interface	44
3.3	Handling Connections	46
3.3.1	Multiple Connections	46
3.3.2	Checking Connections	47
3.3.3	Holding Connections	48
3.3.4	Closing Connections	49
3.4	Caching	49
3.5	Rollback	50
3.5.1	Rollback Mechanism	50
3.5.2	Rollback Configuration	51
3.6	Ordered Queries	52
3.7	Error Handling	53
3.7.1	Exceptions Logging	54
3.7.2	Output Log File	54
4	C++ Parameter Object to Table Mapping	57
4.1	Creating SQL-aware Parameter Container	57
4.1.1	FairRoot Tutorials	57
4.1.2	Parameter Class Ownership	60
4.2	Creating the Object Table	61
4.2.1	Naming a Table	61
4.2.2	Describing a Table	62
4.2.3	Transient Table	64
4.3	Parameter Intrinsic SQL I/O	65
4.3.1	C++/SQL Data Types Mapping	66
4.3.2	Data Representation	66
4.3.3	Data Conversion	67
4.3.4	Single Data types	68
4.3.5	User-Defined Data types	69
4.3.6	Storing Large Object in a Database?	71
4.4	Advanced SQL I/O Features and Optimizations	72
4.4.1	Table Schema Evolution and Context Selection	74
4.4.2	Caching Activation	75
4.4.3	Ordering Rows	76
5	Versioning Management	77
5.1	Validation Table	77
5.2	FairParTSQLIo Responsibilities	79
5.2.1	Run Numbers versus Timestamps	79
5.2.2	Runtime Database and SQL I/O Interface	80
5.3	Templated I/O	83
5.3.1	FairDbWriter Template	83
5.3.2	FairDbReader Template	86
5.4	Condition data	88
5.4.1	Data Members Structure	88
5.4.2	Template Instantiation	89

Contents

iii

5.4.3	Data Encapsulation	90
5.4.4	Storing Condition Data	91
5.4.5	Accessing Condition Data	95
5.4.6	Inactive Cache	96
5.4.7	Steering macros	96

Bibliography

101

Part I

Basic concepts

The Virtual Database

Contents

1.1	Introduction	2
1.2	Problem statement and motivation	3
1.2.1	Files versus database	3
1.3	Database in practice	4
1.3.1	Work overheads	5
1.3.2	Selecting a database	5
1.4	New database technologies	7
1.4.1	The CAP theorem	7
1.4.2	Eventual-consistency	7
1.5	Design	9
1.5.1	Interface as compromise	9
1.5.2	Architecture	10
1.5.3	FairRoot integration	11
1.6	Parameter data types	12
1.7	Mapping data objects to tables	13
1.8	Version management	17
1.8.1	Temporal database	17
1.8.2	FairRoot initialisation scheme	18
1.8.3	Validity time interval	19
1.8.4	Validation basic rules	20
1.8.5	Relational model implementation	21
1.9	Persistency scheme	24
1.9.1	SQL processing	24
1.9.2	Disk cache	25
1.10	Connection Pooling	26

This chapter describes an interface to different database systems, focusing primarily on database applications related to physics data processing. In particular, the use of databases in the accelerator sector, as well as for administrative applications are not covered.

1.1 Introduction

Handling informations related to the status of the detectors at the time of the acquired event is crucial in an experiment. This type of time-varying data are not directly related to a DAQ recorded event and can instead be valid for a set of such events or even change within an event. They are often grouped under the terms of *conditions* data or *parameters*.

The Virtual Database Library aims to provide the necessary tools to store and access non-event time-varying data independently of the Data Base Management System (DBMS) backends. The interface is based on the SQL Server services of the ROOT framework [6], allowing the user to load at run-time the appropriate plugin library in order to communicate with a MySQL [8], a PostgreSQL [9] or either an Oracle server [23]. Traditionnally in the FairRoot framework parameters are managed by the runtime database library taken as legacy code from the Hades analysis software. The runtime database is a parameter manager which can write or initialize parameters to/from ascii or binary ROOT files. The Virtual Database has been integrated to the runtime database library in the FairRoot framework [19] so that it extends its storage capability to RDMS backends in a fully backwards-compatible way without disrupting previous user applications.

1.2 Problem statement and motivation

Motivating a physicist to use a sophisticated database system in his/her data analysis program could be a rather challenging task. Even if changing to a database will make their job easier, most people would rather fall back on old, familiar ways. Indeed, why should a complicated database system be used if, in the FairRoot framework, one can easily initialize parameters using simple ascii or Root binary files?

Overcoming the natural user reluctance to a new database solution begins by showing the general limitations of the file oriented storage solution.

1.2.1 Files versus database

Before comparing, it is useful to start with some definitions:

File: a collection of records or documents dealing with one organisation, person, area or subject i.e manual (paper) files or computer files. Hierarchical storage and organization of files is done by the **file system**, part of the operating system.

Database: a collection of similar records with logical relationships between the records designed to be *shared* by multiple users. Efficient access to information is done by the **database system** not usually part of the operating system.

The most familiar and easy way of storing collection of records is to use a file content. By adopting this *lightweight database model* one usually has to develop a home-grown database relying on third-party persistence and query engine associated to a metadata capable system. As a consequence, this option leads to additional costs in terms of development and maintenance. Furthermore, in the file content approach, one notice that:

- the data is stored as records in regular files.
- the content or file format is arbitrary: any structure is imposed by the author of the file and not by the operating system.
- the usage of multiple files and formats, duplication of information in different files leading to data isolation, redundancy and inconsistency.
- the records usually have a simple structure and fixed number of fields.
- a fast access is only possible by indexing of fields in the records.
- no mechanism exists for relating data between files.

- it is difficult to access and manipulate the data: one needs dedicated programs.
- uncontrolled concurrent accesses by multiple users often leading to inconsistencies i.e two user applications reading a file and updating it at the same time.

Database systems are meant to offer solutions to the above problems occurring when using files to store data. In the database approach :

- **a single repository** of data is maintained that is defined once and then accessed by various users. The repository keeps an *history* of all records (online/offline parameters, experimental setups, etc ...).
- **the database system is self-describing**: the database system contains not only the database itself but also a complete definition of the database structure and constraints. Additionally database systems stores in the catalog the **Meta-data (the data about data)** describing the structure of the primary database.
- **program-data independence**: the structure of data files is stored in the DBMS catalog separately from the access program. It is a big advantage compared to file processing where any changes to the structure of a file may require changing *all programs* that access the file.
- **efficient search / access** to large amount of data with specific properties without caring about the physical storage format
- **sharing of data and multi-user transaction processing**: which allows a set of concurrent users to retrieve from and to update the database. Additionally the concurrency control system within the DBMS guarantees that each transaction is correctly executed or aborted. Controlled concurrent access is essential for performance, especially in distributed computing environments.
- **the system ensures data consistency and integrity**: data integrity is more about the quality of data and may go beyond database management systems. DBMSs provide data consistency tools that can help with data integrity. For example integrity constraints and triggers help ensure that a DBMS does not degrade the integrity of the data that arrives.

With the listing above we have shortly reviewed the benefits of database systems and established how they are superior to the file-oriented data storage systems. One can already get a glimpse of the features of database systems that produce several benefits. Database systems integrate, corporate data, and enable information sharing among the various working groups in an experiment.

But what are the implications when an experiment decides to make the transition from file-oriented systems to database systems?

1.3 Database in practice

When an experiment changes its approach to management of data and adopts database technology, crucial questions immediately arise:

- What are the work overheads?
- Which database to use?

1.3.1 Work overheads

Moving the user application to database systems inevitably means adding overhead:

- Databases are complex systems (client-server application), difficult and time-consuming to design.
- Initial learning of Structured Query Language (SQL [11]) is mandatory for database applications. SQL is a declarative language which does not specify the implementation details of database operations and uses query optimization to determine an efficient data access plan. SQL is universally accepted as the basis of all leading RDBMS systems today and is the actually dominant database language.
- Additional learning and training period is required for the users. One should define who will have the responsibility to perform data entry and how the data entry will be performed.
- Further effort in training for database developers is needed to significantly reduce the amount of effort required to solve design or key implementations mistakes.
- *Garbage in - garbage out* : a damage done to a central database will affect the whole collaboration and usually the recovery is difficult.
- Substantial hardware and software start-up costs are expected.

1.3.2 Selecting a database

Choosing a database can be today a rather challenging task considering the impressive amount of commercial, non commercial, relational or non-relational database systems.

With the recent explosion in database technologies it is even more difficult to choose the proper system (Figure 1.1). Nevertheless one can distinguish two main groups: the traditional *all-purpose* relational (SQL) database (DB2 [21], Oracle, MySQL,...) which are bullet-proof reliable systems but facing limited scaling and the *special-purpose* NoSQL non-relational (NoSQL [24]) databases (Hadoop [20], CouchDB [10], mongoDB [12], ...) known to be massively scalable and distributed but having lower availability, reliability and a weaker consolidation on the market. Theoretically , *all-purpose* databases provide all features to develop any data driven application: a powerful query language SQL which can be used to update and query data even via very complex analytical queries, and an expressive data model where most data modeling can be served by the relational model. It is definitely this technology that one should consider in the first place. The NoSQL database should be seen as complementary solution to cope with massively high query loads in a distributed environment where *all-purpose* database may lead to problems.

One can try to reduce the list of possible database candidates introducing general criteria such as efficiency, robustness, security, C++ interface access, concurrent writes, portability of data, cheapness.

Unfortunately none of the systems excels in all categories, although the commercial systems fared best in their feature set and clearly worst in terms of costs.

Further more experiment specific requirements could be added to the general criteria for database selection.

- **What data volume need to be stored and how long?** Quantitatively, one can investigate the present situation for parameters storage volumes at LHC. For example ATLAS [15] has the larger parameter database mirroring the high number of



Figure 1.1: **Which database to choose ?** This figure shows only a non-exhaustive list of commercial, non-commercial, relational, non-relational database systems.

channels and the corresponding complexity of its software handling: presently 1.2 TB parameters data including indices and increasing 0.7 TB per year compared to both CMS [16] (200 GB, increasing 20 GB per year) and LHCb [17](2 GB in SQLite, increasing 0.6 GB per year). It should be noted that these numbers only include the parameters stored using relational databases, but part of the payload is stored in referenced external files, both in LHCb (magnetic field maps) and in ATLAS (POOL files).

- **What tasks will be perform and how often will data be modified?** The requirements are different when a database system should be use for online event selection or offline analysis. In particular for the online case the database technology has to be evaluated in the context of distributed computing where many programs very often will access/update parameters.

One could also looks what other experiments with similar computing environment have experienced by using database systems. For example let's take the experiments at LHC-CERN [7]. On one hand, the LHC experience shows that relational database technology can be sucessfully used to store parameters even up to a 1 TB data volume. Database cluster solution together with a dedicated deployment strategy (Frontier and Squid server) are used for computing intensive analysis jobs. On the other hand, the FAIR computing environment will have to face a situation where all raw data will be streamed into the data acquisition and need to be filtered before being recorded to tape. This implies that event association must be performed in software online which go beyond the normal event-by-event processing of the traditionnal trigger based experiments such as LHC experiments. This unique data aquisition concept of the FAIR experiments further constrain the evaluation of database technology within the context of distributed computing where RDBMS databases might not deliver an optimal solution.

With the FairRoot Virtual Database, the user will be able to integrate different database technologies, prototype many ideas, measure and compare. This process is inevitable in order to select which database systems meet the best the computing requirements of an

experiment.

1.4 New database technologies

1.4.1 The CAP theorem

In the context of distributed computing a fundamental constraint comes from the **Consistency, Availability, Partition Tolerance (CAP) theorem**. In 2000 Eric Brewer [5] proposed the idea that in a distributed system you can not continually maintain perfect consistency, availability and partition tolerance simultaneously. **CAP** is defined as:

Consistency: all nodes see the same data at the same time.

Availability: a guarantee that every request receives a response about whether it was successful or failed.

Partition Tolerance: the systems continues to operate despite arbitrary message loss.

The CAP theorem states¹ that you cannot simultaneously have all three; you must make tradeoffs among them (Figure 1.2). The CAP theorem relates these tradeoffs that exists among **Consistency**, **Availability**, and **Partition tolerance** for systems that provide distributed access to data. In this terminology, **Consistent** means that any part of the overall system, if and when it responds to a request for data, provides precisely the correct data. **Available**, on the other hand, means that all parts of the overall system are *always up*, and any component will promptly provide an answer to any request. Finally, **Partition tolerant** means that the system continues to function in the face of network disruption (or partition).

The CAP theorem says that while it is possible for a system for distributed data access to possess two of the three properties of consistency, availability, and partition tolerance, it is outright impossible to achieve all three simultaneously. Said another way, when the network goes down, implying the system has no choice but to become partition-tolerant, you must give up either consistency or availability. Either you cannot get to your data ("the network is down!"), or you run the risk of causing inconsistencies ("someone changed the file i had opened and i lost my work!"). People designing networks and distributed applications are familiar with these tradeoffs, and often balance them skillfully in working through a problem.

Anyone familiar with databases will know the acronym ACID, which outlines the fundamental elements of transactions: atomicity, consistency, isolation and durability. Together, these qualities define the basics of any transaction. With the **CAP theorem** it became clear that in order to deliver scalability it might be necessary to relax or redefine some of these qualities, in particular consistency and durability [22]. Complete consistency in a distributed environment requires a great deal of communication involving locks, which force systems to wait on each other before proceeding to mutate shared data. Even in cases where multiple systems are generally not operating on the same piece of data, there is a great deal of overhead that prevents ASIC compliant database system like SQL relational database systems from scaling.

1.4.2 Eventual-consistency

To address this limitation, a possible solution is to consider a storage system that relax the notion of complete consistency to something called *eventual consistency*. This allows

¹The CAP theorem began as a conjecture made by Eric Brewer in 2000 and has been formally proven by Seth Gilbert and Nancy Lynch of MIT in 2002, rendering it a theorem.

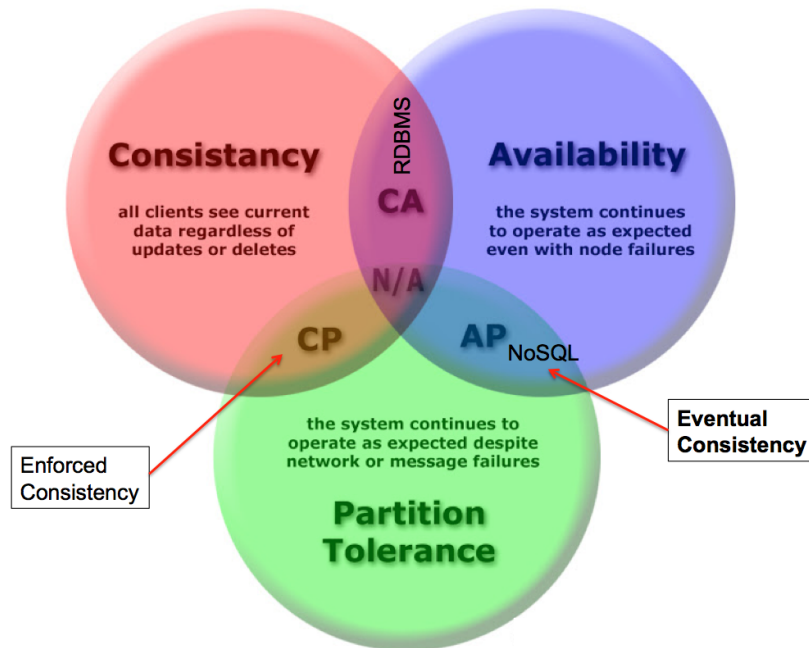


Figure 1.2: The CAP theorem relates the tradeoffs that exists among **Consistency**, **Availability**, and **Partition tolerance (CAP)** for systems that provide distributed access to data (cf. [5] slide 12).

each system to make updates to data and learn of other updates made by other systems within a short period of time, without being consistent at all times. This *eventual-consistent* database technology is often called NoSQL database technology where the “no” stands for “not-only” as the goal is not to reject SQL but, rather, to compensate for the technical limitations shared by the majority of relational database implementations.

The technologies most frequently mentioned in the discussion about possible alternatives to relational database systems are HBase, Cassandra, MongoDB, CouchDB and Membase. The perceived advantages of such systems include simplicity, scalability, efficient data distribution and possibly better performance. A careful evaluation of NoSQL technologies is still needed in order to understand and measure any advantages of this technology over the traditional RDBMSs. In particular, it should assess the relevance to conditions data of those features of NoSQL technologies that are not available in traditional RDBMSs (e.g. **MapReduce**² [13]).

In order to be able to perform such important tests, the data model of the FairRoot database framework should be **as simple as possible** to be easily implemented in a SQL-based system as well as a Key-Value model in a NoSQL system. A complex relational database model will certainly not fit well with NoSQL technologies. For example if, from the beginning, the relational data model uses extensively joins of tables, the complexity of the query processor

²**MapReduce** is the programming paradigm of many key-value database systems that allows for massive scalability across hundreds or thousands of servers in a cluster. The term MapReduce actually refers to two separate and distinct tasks that a program perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual entities are broken down into tuples in form of key-value pairs. The reduce job takes the output from a map as input and combines those data tuples into a smaller set of the tuple. As the sequence of the name MapReduce implies, the reduce job is always performed after the map jobs.

will increase and the query time will consequently increase especially in distributed systems. Additionally simplicity as a general design guideline is always an advantage: in the lessons learned during the development phase of **BigTable** from *Google, Inc* the authors quoted [14]:

In the process of designing, implementing, maintaining and supporting BigTable, the most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging.

Following this statement, in the FairRoot virtual database design, the relational model is kept simple so that it can be ported easily to a simple *Key-Value* [3] similar to NoSQL database storage model later on.

1.5 Design

Which database system, relational or non-relational, will meet the requirements of a specific experiment? Who can do the necessary measurement/ performance tests to help defining the system? Alarm bells are already ringing: are then the proposed database solutions compatible with the manpower resources that will be available to eventually support them ? Unfortunately, the answer has to be related to the fact that the number of true experts in the dark arts of database technology are closer in number to those in the early day of relativity. Even for some FAIR experiments to find such experts would be nearly hopeless.

On top of all these facts, one should add the cost of administering the databases for the experiments, which is definitely significant: the use of database system usually implies a heavy infrastructure, which not even the experts in the experiments can fully control without the support of their IT colleagues.

Anything that can be done to reduce this effort - over and above the reduction in support load that would come from an adequate design and implementation - would then be welcome.

1.5.1 Interface as compromise

Having noted all these points as guidelines for the development of the parameter persistency system, one should first and foremost ensure that the new system do not disrupt the present production service that the FairRoot framework deliver.

This means that the new database system should be built on top of the existing *Runtime Database* [4] library which provide the general API for file-oriented parameters storage (ASCII or ROOT binary file format). The new persistency system will then extend the storage system to traditional relational or non-relational database system keeping the former API to ensure backward-compatibility.

Additionally, as mentioned before, one should keep in mind that technology in the area of database evolves very rapidly, and what looked like promising technologies 10 years ago may not be so now, and one must ensure that the parameter persistency system in the FairRoot framework is agile and responsive to be able to adapt to a new coming trend and, that way, remains current and relevant.

These aspects describing what the usage of a database implies for an experiment should be taken into account while designing the database system for the FairRoot framework. This

brings to the concept of a FairRoot Virtual Database, natural approach when one want to integrate different database technologies, prototype many ideas, measure and compare.

Gaining acceptance for database solutions starts all the way back at the drawing board: the complexity of database usage should be reduced to a possible minimum while a large panel of different database technologies should be supported. Following the concept of the Virtual Monte Carlo [18]³ the Virtual Database library is an attempt to ease the transition from flat files based parameter storage to DBMS storage.

With the FairRoot Virtual Database, the user code follow a well-defined and uniform API for database manipulation and is then defined independently of a specific database system. It can then be run with all supported database engines which are presently MySQL, Oracle and PostgreSQL.

1.5.2 Architecture

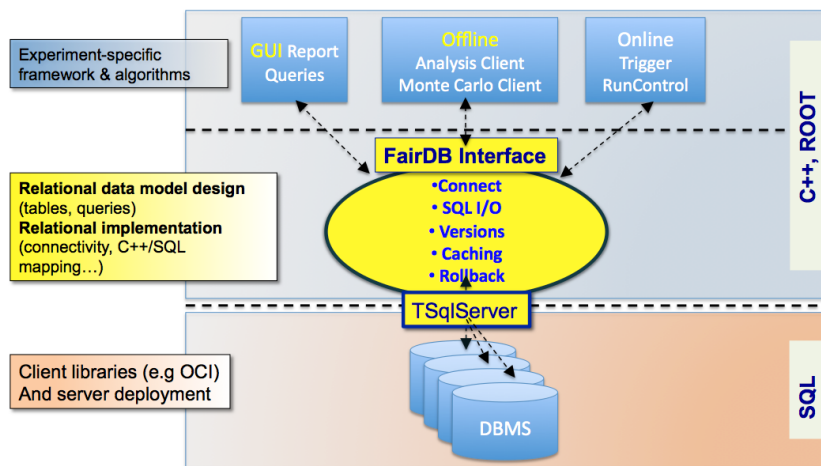


Figure 1.3: The Virtual Database library (FairDb) scheme.

The FairRoot Virtual Database library (FairDb) separates the connectivity to the DBMS into a *front-end* and a *back-end*. The *front-end* delivers the relational data model design and implementation (tables, query algorithms, version management ...) together with the relational abstraction layer responsible for database connectivity, authentication and C++/SQL input-output mapping. FairRoot experiment-specific applications use only the exposed *front-end* API delivered by the system to interact with the database system.

The *back-end* defines the facilities that communicate with specific DBMS (MySQL, Oracle, PostgreSQL, etc.) using appropriate client libraries (e.g MySQL C API, Oracle Call Interface ...) and provided by low-level *devices drivers* that get invoked automatically and dynamically loaded into the ROOT session at start time using the regular ROOT plu-

³The concept of Virtual Monte Carlo (VMC) has been developed by the ALICE Software Project to allow different Monte Carlo simulation programs to run without changing the user code, such as the geometry definition, the detector response or input and output formats. The strategy followed by the ALICE collaboration was from the beginning to develop a simulation framework that would allow a smooth transition from the currently used transport code GEANT3 to more complex ones Geant4 and Fluka. FairRoot as a simulation framework is entirely based on VMC.

gin mechanism. The *back-end* layer uses the `TSQLServer` services of the `ROOT` library which provides an abstract interface allowing SQL queries execution and manipulation of retrieved results sets. Presently `TSQLServer` supports the following relational database backends: MySQL, Oracle, PostgreSQL and SQLite.

The Figure 1.3 shows the general scheme. The software stack for handling parameter data can be seen as composed of three separate layers. The bottom layer or *back-end* layer, closest to data storage and service deployment, includes the server and client components of the database management systems where data are physically store (MySQL, Oracle or PostgreSQL, provided by external vendors). On the layer above, the *front-end* layer, sits the Virtual Database library which experiments use to decouple these low-level relational backends from the parameter database applications. Shared by all different users within an experiment, it also contains the design and implementation of the relational data model for parameter data including all relevant tables and queries. Finally the last layer, experiment-specific, includes the generic software framework and the various algorithms used by the many detector and physics groups in each experiment.

1.5.3 FairRoot integration

The parameters data within the FairRoot framework is handled habitually by the runtime database which mainly consists of a singleton manager class `FairRuntimeDB` holding both a list of parameters and the corresponding interfaces for accessing and storing these parameters. The general interface class `FairParIO` give the user the possibility store and access

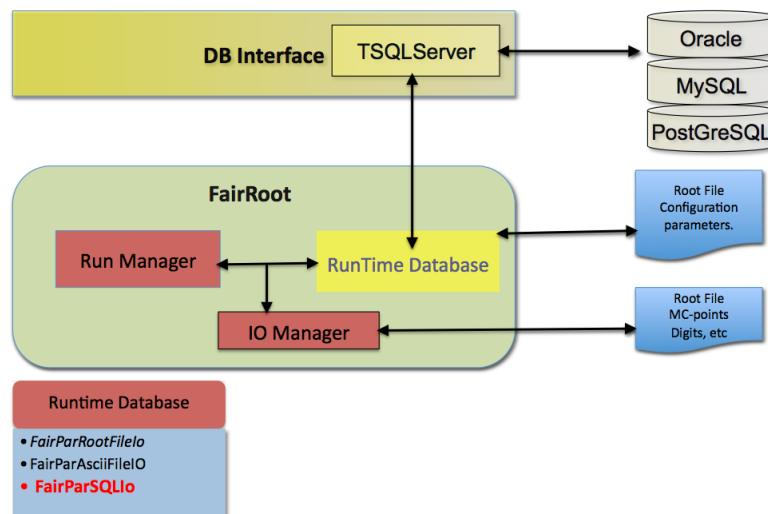


Figure 1.4: The Virtual Database scheme within FairRoot.

conditions data from two main sources:

ASCII file this mode is intended for an easy and convenient access to parameters, mostly used for prototyping and testing purposes.

ROOT file this mode is a convenient way to have local copies of the parameter data using the binary object streaming mechanism automatically provided by the `ROOT`

framework. Additionally to reduce possible streaming inconsistencies and reinforce backward compatibility, the ROOT object I/O system provides a Class Schema Evolution which can handle the case when a data member is added or removed from a class in a manual or automatic way.

The data streamed from ASCII files or ROOT binary file is then mapped in memory to specific parameter container objects which are stored using the ROOT polymorphic collection `TList`. All parameters class are subclass of `FairParSet` which deliver the API for a *generic parameter set*. Each parameter container in the list is identified by a name. One can retrieve a container via its name with the function `FairRuntimeDB::getContainer()` and add new container to the list with the function `FairRuntimeDB::addContainer()`. Additionally the `FairRuntimeDB` manager class connect the input and output streams using the abstract class `FairParIO` holding an array of `FairDetParIO` objects. The `FairDetParIO` abstract class defines the generic interface used actually to read and write the parameter containers consisting of two main functions:

`FairDetParIO::init(FairParSet *par)` The data streamed is mapped in-memory to the `FairParSet` parameter object pointed by `*par`

`FairDetParIO::write(FairParSet *par)` the data members of the class pointed by `*par` is streamed to either an ASCII file or a binary ROOT file using the ROOT persistency mechanism.

The Virtual Database extends the FairRoot runtime database I/O streams to relational database backends adding SQL I/O mechanism to the generic parameter set class `HParSet` so that it can hold data retrieved from a database. It also defines the C++/SQL mapping with the I/O interfaces `FairParTSQLIO` and `FairDetParTSQLIO` as subclasses of `FairParIO` and `FairDetParIO` respectively.

The `FairParTSQLIO` class is a singleton manager class that uses the Virtual Database general services to

- connect to one database or multiple databases in a ROOT session
- communicate with the underlying database using SQL statements.
- translate if necessary the SQL statement to the specific SQL of the choosen database system.
- execute that SQL queries to the underlying database
- view and modify the SQL resulting records.

1.6 Parameter data types

The non-event ⁴ data to be stored in a database can be classified into different groups:

Construction data The data gathered from both the production process and the produced items during the construction of the detector. Construction data need to be stored in a dedicated database, often called *hardware database*, for the lifetime of the experiment in order to trace back production inconsistencies or/and errors. The construction data and its corresponding storage is not covered by this document.

⁴The data needed to set up the detector but not produced by it.

Configuration data The data needed to bring the detector into a proper running mode i.e voltage settings of power supplies as well as programmable parameters for front ends electronics etc...

Calibration data The data describing the calibration and the alignment of the individual components of the different detectors or sub-detectors. Usually these quantities i.e drift velocities, displacement constants, etc... are evaluated by running dedicated offline algorithms. However in a free-streaming DAQ environment they have to be evaluated online since they are needed by the reconstruction. In such a mode, they will have to be stored in a dedicated online database.

Parameter data The data that describes the state of any detector subsystem and how it responds to the passage of particles through it. They include data quality indicators such as bad channel lists, a variety of detector settings (such as pedestals) including the geometry and material definitions.

Algorithm data The data used to control the way an algorithm operates. This includes for example software switches to control data flow and tunable cuts for selection.

It is the responsibility of the database interface to provide the framework in which all different type of parameter data can be model and new types can be added with little effort. Except may be for the case of the Construction Data, all types of data must be described *at least* by a version and the time validity interval information corresponding to the set of data for which they are measured. For this reason, it is also the responsibility of the virtual database to provide a versioning management for the description and the access to such data including both insertion and retrieval.

1.7 Mapping data objects to tables

Conceptually, the virtual database handles data as table i.e in a two-dimensional data structure with cells organized in rows and columns. Eventhough tables are the basic building blocks of a relational database model, data organisation in terms of rows and columns is also used in non-relational databases. Indeed in *key-value* based storage, data is simply organized as persistent multidimensional map indexed by a row key, a column key and usually a timestamp. Therefore defining a data model in terms of rows and column as the basic data organisation for the virtual database does not lack of generality.

Because relational data is not generally expressed in flat data models, but rather, grouped into modules that are connected through relationships, it is often difficult to normalize relational information into a flat object. The virtual database eases the transition object-to-relational database by implementing an automatic **mapping**⁵ procedure according to the following rules:

- **Class** definition map to table definition
- **Columns** are the physical equivalent of attributes.
- **Rows** are the physical equivalent of object instances

⁵ The terminology mapping refers to the reversible deconstruction of an arbitrary set of C++ data structures (objects) to one or many relational tables in a SQL-based database system. The table data model can then be used to fill an equivalent structure with the proper parameter data in another program context.

- **Object Identifier:** each record (rows or unit block of rows) on a table is uniquely identified. The purpose of the unique identifier is to act as the *primary key* on the table where it is defined and to be referenced as the *foreign key* by other related tables. The Object Identifier is implemented as a sequential ⁶ number key which is independent of all other columns and insulates the database relationships from changes in data values and guarantees uniqueness.
- **Relationships** are not implemented in the relational model i.e retrieval of data from multiple database tables simultaneously to reconstitute the object structure is not supported. This design choice reduces internal query processing complexity. ⁷

The data storage used in the virtual database is a simple row-oriented storage which means that a table is stored as a sequence of records, each of which contains the fields of one row. When the user requests data from a table, the virtual database collect rows of data from the appropriate table and map the data to the corresponding data members of the target user-defined parameter class.

From the perspective of the virtual database, each parameter class definition corresponds to a dedicated main table structure that hold the bulk of data. The unique identification of a row in the primary table is achieved by the definition of a `Seq_Id` column and a `Row_Id` column. The data itself consists of additional user specified column fields corresponding to each persistent data member type in the parameter class definition. The `Row_Id`, in conjunction with a unique `Seq_Id` allowed the pair to be indexed as a *primary key*, which require uniqueness, for integrity purposes.

Eventhough the object to table mapping can be user defined, the virtual database provides 2 main types of data representation which are illustrated in Figure 1.5 and in Figure 1.6.

One-to-One mapping In this case the target parameter class is a simple class or is related through inheritance 1.5. Through the virtual database interface, to both the base class and derived class a corresponding relational table is created and then filled with updated values of the parameters.

The object data mapping including the object relationships is then reconstituted by polymorphic IO functions `Store()` and `Fill()`, acting like the `ROOT IO TObject::Streamer()` virtual functions. When in another program the same object structure need to be filled or updated a single row is retrieved using an appropriate context.

This is usually the case for Algorithm Configuration data which follow most of the time a One-to-One mapping; even if multiple configurations are possible, only one can be selected at a time. More generally the other types of data like detector related parameter data is almost never simple.

One-to-Many mapping In this case the target parameter class uses aggregates and is composite 1.6. As for the one-to-one mapping the for both container class and aggregates class a corresponding table is created and the object data mapping including the object relationships is then ensured by polymorphic IO functions `Store()` and `Fill()`. When in another program the same object structure need to be filled or updated multiple rows are retrieved using an appropriate context i.e *primary reference key*. Each row maps exactly to a single main or aggregate object

⁶ The sequential number is implemented as a column defined with `AUTO_INCREMENT` in MySQL, with `serial` type in a PostgreSQL and with a `SEQUENCE` in ORACLE

⁷ for example by avoiding the query-time `JOIN` which obviously increase complexity of the query processor, especially in distributed systems.

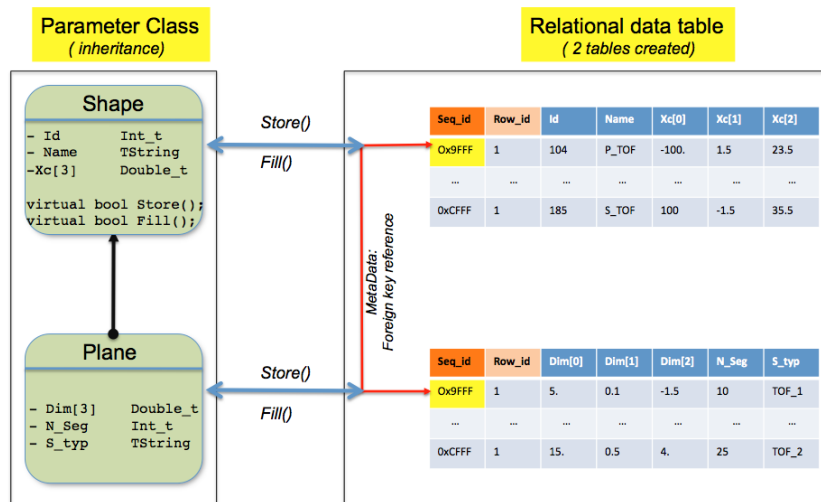


Figure 1.5: **One-to-One mapping** with the target parameter class being related through inheritance. Through the virtual database interface, to both the base class and derived class a corresponding relational table is created and used. The object data mapping including the object relationships is then ensured by polymorphic IO functions `Store()` and `Fill()`

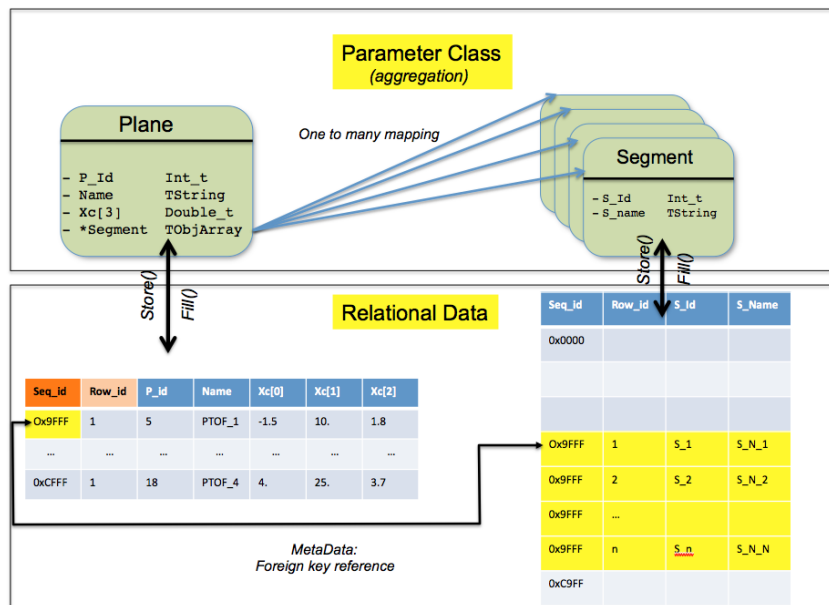


Figure 1.6: **One-to-many mapping** with the target parameter class defining aggregates. As for the one-to-one mapping the for both container class and aggregates class a corresponding table is created and the object data mapping including the object relationships is then ensured by polymorphic IO functions `Store()` and `Fill()`.

and the request retrieves data for a complete set of aggregate objects. For example a request for scintillators positions in a TOF wall will produce a set of rows, one for each scintillators parameter object.

Incremental One-to-Many mapping Another form of composite persistent object is foreseen by the interface depending on the way new data is added to the database. If the parameters for the entire system to be described is written as a whole chunk (Figure 1.6: many rows are retrieved for one unique `Seq_id`) the composite object can be reconstituted only as a whole logical block. This could be the case for a table that describes the way silicon wafers map to electronics channels: the complete object will be written and reconstituted as a single unit.

But the composite object can also be written piecewise i.e. in smaller chunks, every small chunk units having probably different sequence identifier. The use-case is for example when a partial calibration is done and only a small subset of calibration parameter is updated. In this case only the updated part of the data will be added to the corresponding table with a new `Seq_id` the rest staying unchanged. The object to table mapping is then **incremental** i.e. later on if the composite object is requested to be updated by the program, only the newly updated calibration parameters will be streamed thus reducing the IO. The tasks of the interface during the mapping is to reassemble all small chunks to reconstitute integrally the composite object.

The mapping process can be twofold:

- it can be complete so that the number of small data units is constant which is normally the case if one describes a set of parameters that are defined permanently i.e. position of the TOF planes.
- it can be sparse if the number of small data units is variable i.e. segmentation of a TOF plane changing according to a requested new granularity.

FairDb provides a programmatic and uniform API to access ROOT parameter object regardless of its internal relational representation. Like for the Java SQL interface system (JDBC), each data retrieval produces a pointer giving read access to a results table, which corresponds to a subset of the underlying database table. Each row of the results table is an object, the type of which is user defined and table-specific.

Through the interface, the data access mechanism from one row hides the way the database table is organised. So changes to the physical layout of a database table should never effect the memory mapped user defined parameter object but only effect the internal table row representation with FairDb⁸. If for some reason, the writing of parameter object fails, then in the resulting table no new row will be added, otherwise it will have a single row for a simple structure and more than one row for a composite structure. Using the `TSQServer` services, the user can manipulate the result set after a request: he can ask how many rows the table has and can directly access any of them. The rows are physically ordered in the table reflecting the way the data was originally written.

In order to further optimize the retrieval mechanism, it is possible to create independently of the physical position of the row an additional index. In the **Incremental** one-to-many mapping the table layout reflects the way the small data chunks are written. The interface incremental mechanism replaces individual data-units as their validity expires. Unfortunately this means that the physical layout may not be optimized for access. To cope with

⁸ Retrieval of data from multiple database tables simultaneously is not supported. A single request only accesses a single table.

this parameter objects in the runtime database, which all inherit from the base parameter class `FairParSet` can always return a table index, if the physical row ordering is not an access-optimized one. Corresponding rows in the table can then be accessed faster using this index ⁹.

1.8 Version management

The main purpose of using a database is to keep a complete history of all written records in order to be able to easily answer, for example, the following question:

What was the position the TOF wall for the october 2006 experimental setup?

Unfortunately the above simple question can not be answered with a conventional relational database since data versioning is not directly supported by the database system. Traditional relational databases are often called *snapshot* databases as they only show the relationship of tables at one moment in time. When a change is made to a data table or to its relation, it is updated and its state is changed. All past states are discarded and cannot be retrieved anymore. In many situations, the loss of past information is not a problem as old information will not be needed again. However condition data is a typical time-varying data type i.e retroactive, current information is desired as well as when each piece of data is valid.

For this reason the version management has to be implemented as a add-on service on top of the database system. In FairDb the version management is simply based on time since the variable time is monotonic and has a sensible ordering ¹⁰. Additionally time can also serve to unify information which comes independently in a asynchronous data acquisition system. For example a run number 1 for the time of flight detector should not automatically correspond to run 1 for the tracker detector. In this case, the event correlation between detectors can be easily done using time ¹¹.

1.8.1 Temporal database

Time or more precisely time-slice is a natural detector buffer identifier in a free data streaming data acquisition. The only technical restriction is that time needs to be **UTC defined** ¹² [1] in order to avoid the two time a year daylight saving time shifts which can leave holes or add ambiguity challenging the data integrity of the database. In FairDb the timestamp database field is structured to present itself to the user in standard **ISO 8601 format YYYY-MM-DD hh:mm:ss**

Conceptually, in a conventional database data is organized in two-dimensional space (attribute, tuple). Adding time-based versioning means that time is added as a third dimension leading to a three dimensional model (attribute, tuple, time).

⁹ A standard way to improve the performance of `SELECT` operations is to create indexes on one or more the columns that are testes in the query process. The index entries acts like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the `WHERE` clause i.e for a 1000-rows table one can reach a factor 100 times faster than pure sequential access.

¹⁰in order to avoid the two time a year daylight saving time shifts which can leave holes or add ambiguity challenging the data integrity of the database. The run number usually used for initialisation scheme is not a good candidate since parameter can change independent of run boundaries. For example the temperature on front-end electronic board will certainly change within a run as it mostly depends on the beam spill structure.

¹¹ Time-of-flight being negligible at the quantization of the database using only timestamping (`TIMESTAMP`) of a second precision.

¹² **Coordinated Universal Time (UTC)** is the primary time standard by which the world regulates clocks and time

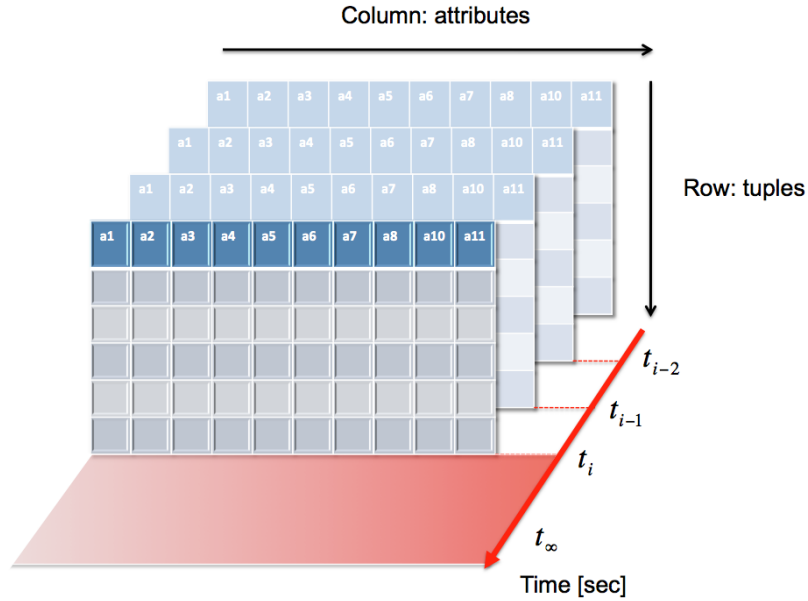


Figure 1.7: Time-based version management can be interpreted as an extension of the conventional two-dimensional database data where time is added as third dimension.

The Figure 1.7 illustrates the corresponding database data model which can be thought of as a cube with the following axis: column-attribute dimension, row-tuple dimension and time. Each surface (row,column) corresponds to a point in time. All the valid tuple information from each record at that time point are contained in the surface. However for the concrete implementation of the version management, the time dimension is discretized allowing the definition of *interval of validity*: a parameter data update at time t_i will always correspond to a entry related to a validity interval $[t_i, t_\infty[$. The theoretical infinite time t_∞ is in fact limited to the Unix operating system time which is stored as a signed 32-bit integer in a dedicated structure (`time_t`). The Unix time is interpreted as the number of seconds since 1970-01-01 00:00:00 UTC which consequently give the furthest time that can be represented this way to be defined until 2038-01-19 03:14:07 UTC.

1.8.2 FairRoot initialisation scheme

In FairRoot the initialisation scheme is traditionally based on a generated *run number* which uniquely identifies a sequence of simulated or real events assumed to share the same experimental conditions. The Figure 1.8 illustrates the FairRoot initialisation scheme handled by the runtime database system FairRuntimeDB: every new data file input will trigger a algorithm initialisation/ reinitialisation (`FairTask::Init()` / `FairTask::ReInit()`) which will uses the `RunId` in order to update all relevant parameters for the current run. In principle, any of the parameter data initialised by the FairRoot runtime database system could depend on the current run being processed. Clearly other parameter data types, for example parameters describing detectors, such as calibration constants, will change with time and it is necessary to retrieve the right ones for the current event or even within an event.

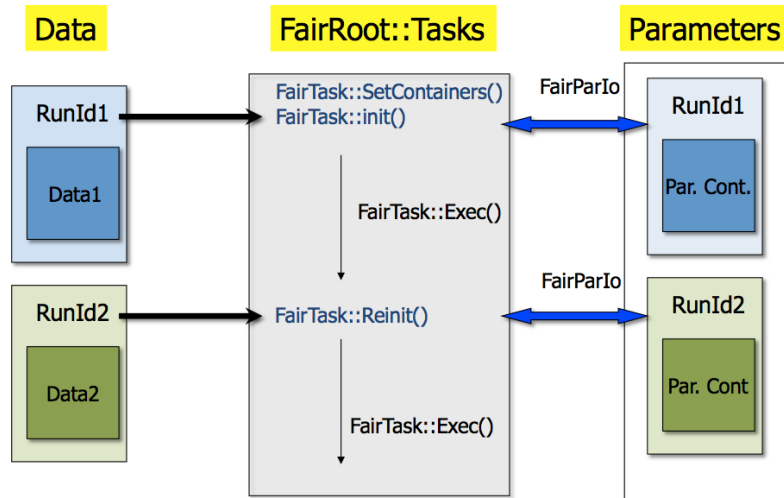


Figure 1.8: FairRoot initialisation scheme based on unique run numbers for simulation or real data streams.

1.8.3 Validity time interval

As already mentioned above, the virtual database copes with this problem by using time instead of run numbers for the version management. All `RunId` are interpreted internally as timestamp i.e all events in a run will have the same associated timestamp. Every time a new event is read within the FairRoot framework, the associated `RunId` is read and the data can be retrieved from the database using the correspondance `RunId` \Leftrightarrow `TimeStamp`. Furthermore any parameter update done for one event or within an event will be timestamped with a new time $\tau \in [t_1, t_2[$ where t_1 and t_2 could be the start times of `run#1` and `run#2` respectively. As shown in Figure 1.9, any query processed with $t_i \geq \tau$ will trigger in the virtual database a retrieval of the best parameter data. This is one by picking the one which has the latest creation date time.

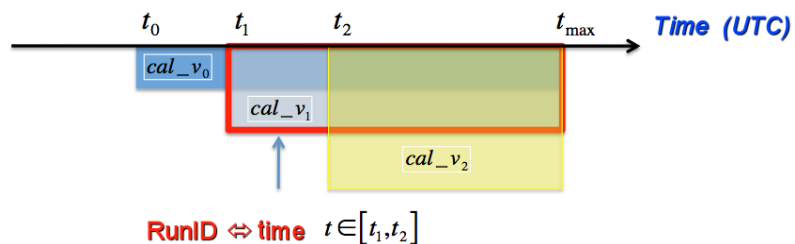


Figure 1.9: In the time-based version management, any query processed with $t \in [t_1, t_2[$ will trigger in the virtual database a retrieval of the corresponding timestamped parameter data `cal_v1`

1.8.4 Validation basic rules

The version management basic rule is that any parameter writing transaction through the virtual database follows a strict write-once policy. No parameter stored in the database can be deleted later on. If another version of the parameter is written it just append to the existing ones. Following this basic append-only rule is a standard way of reinforcing the data integrity in the database.

Another important rule is that any data set whose creation time is later will be better and the system will automatically trim its validation range so as not to overlap it. It happens in memory and results in the creation of an effective validation range, the database itself being untouched i.e no additional check in the database is needed for this data set until the effective validation range has expired. This trimming mechanism also applied in a pool of connected multiple databases. Since the trimming mechanism happens in the virtual database system itself, one can imagine that in a running session with mutiple database connections, data sets in higher priority databases trimming the ones having a lower priority.

Nevertheless the virtual database does not assume that parameter data from earlier runs are created before parameter from later runs. This would be a unfortunate limitation: for instance when improving calibration constants, it is often necessary to recalibrate former runs before going back and applying/fixing earlier ones. Simply using a timestamp when the parameters were created would mean that the earlier run's parameters would take priority over any later run's parameter they overlapped. To allow parameters to be created independently of the order the virtual database provide a mechanism that deduces the best creation date.

In practice, the three dimensional model (attribute, row , time) for the version manage-

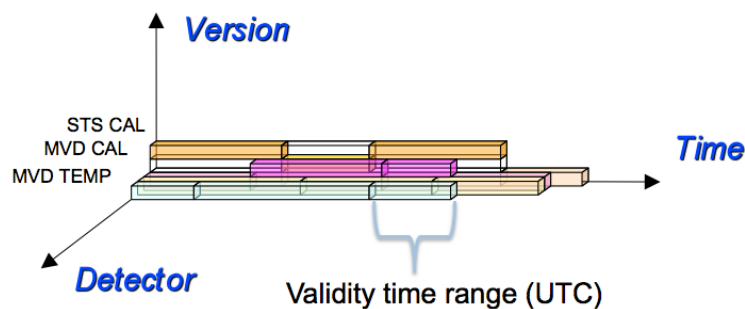


Figure 1.10: Multi-dimensional version mangement. The space coordinates (time:timestamp, data versions:integer, detector type:string) are used to locate the physical data in the database.

ment can benefits from further dimensional extensions. Figure 1.10 illustrates a version management based not only on additional *time* dimension but also on *data versions* and *detector types* dimensions. The set of add-on dimensions put on top of the original two-dimensional (attribute, row) database model creates a so-called *search sensitive context*.

This *context* constitutes the **metadata**¹³ which determines where the data table is located within the database.

1.8.5 Relational model implementation

The virtual database separates the metadata table (logical data) from the data table (physical data). Separating logical to physical data is a common practice not only for Distributed Storage Systems¹⁴ but also for Parallel File Systems¹⁵.

The virtual database use a two-level hierarchy to store the physical data which is illustrated in Figure 1.11. First, the metadata which locates the data is stored through an auxiliary table. The rows in the metadata table represents a *context object* (**Context** class) which contains the data description in terms of

- **Interval of Validity:** the date and the time of stored element in the database
- **User defined data description:** the detector type, the data type, the data model (simple or composite) and eventually a version number

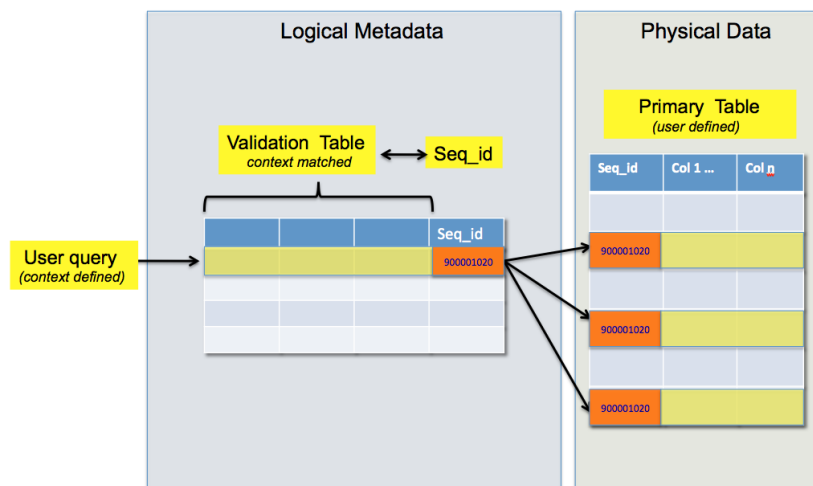


Figure 1.11: Data Table location hierarchy.

Second, when the user wants to retrieve data, the system requires the data description i.e. a *context object* to be provided. With the data *context* information, the system selects the correct set of **Seq_id** values and uses them as keys to retrieve the corresponding data in the primary table. This mechanism can be thought of as a relational implementation of the

¹³ **Metadata** stands for structured information that describes, explains, locates or otherwise makes it easier to retrieve, use or manage an information resource. **Metadata** is often called data about data or information about information. The term metadata is used differently in different communities. In our case, metadata describes a resource for purposes of data location and identification. It includes different descriptive data items defining a multi-dimensional space which aids in data retrieval. More precisely it is called *descriptive metadata*.

¹⁴ The BigTable Data Model from *Google, Inc.* uses separate Metadata tablets connected to multiple User Data Tables.

¹⁵ The Lustre File System splits Metadata Servers and multiple Data Servers. Both are then interconnected through network or directly.

key-store based data access in a multi-dimensional STL map [2].

This two-level hierarchy data access corresponds to a two-table structure¹⁶ for each type of data in a relational data model: a main table to hold the user data payload and a validation metadata table to map the context information (timestamp,detector, etc.) to the single element `Seq_id` primary keys for a set of rows in the primary table.

1.8.5.1 Data payload table

Fields:	Types:	Keys:	Null	Specs
<code>Seq_id</code>	<code>int</code>	PRI	no	<code>auto_increment</code>
<code>Row_id</code>	<code>int</code>	PRI	no	
<code>data_1</code>	<code>user_defined</code>			
<code>data_2</code>	<code>user_defined</code>			
...
...

Table 1.1: The main data table model consisting of locators columns `Seq_id` and `Row_id` followed by one or more user-defined data items columns.

The table 2.1 shows the structure of the primary data table. It consists of multiple primary keys columns (`Seq_id`, `Row_id`), and one or more user-defined columns for the data payload. In this table more than one row can have the same key `Seq_id`. Key-based lookups are done uniquely on the `Seq_id` column. Any queries on the primary table connects all entries matching context-based selections i.e a selection of one or more `Seq_id` values¹⁷.

1.8.5.2 Validation table

The Validation Tables holds the metadata necessary to quickly locate the payload tables in the system. The Figure 1.2 shows the structure of the metadata table. It consists of multiple time dimensions which translates either to timestamps or time intervals and data model and type identifiers. All data stored in the database will automatically be tagged using this information aggregated into the so called *context* represented by the `ValContext` class.

$[t_{start}, t_{end}]$: defines the **Interval of Validity (IoV)** i.e the time for which the `Seq_id` is valid. The time t_{end} is excluded from the interval to avoid ambiguity or problems linked to adjacent intervals. When defining a *context* the user can either use single-sided intervals i.e those which extends from a start time off to an *infinite time* ($[t_{start}, t_{\infty}]$) or limited time range intervals ($[t_1, t_2]$) which allow to put entries for a limited time period. This is relevant for the overall IO performance since it avoid repeated data¹⁸. In a composite data model, multiple rows in the primary table can

¹⁶ Splitting the metadata table from the data payload table allow the system to avoid duplicating validity information.

¹⁷ Query-time optimization is achieved by simply indexing the `Seq_id` column. The `Row_id` in conjunction with the `Seq_id` allow the key-duplet to be indexed as unique *primary key* for data integrity purpose.

¹⁸ Being able to patch limited time ranges reduces data redundancy. Let suppose one wants to mask out few channels for a limited time period without duplicating the whole composite object a second time. In

Fields:	Types:	Keys:	Null	Specs	Default
Seq_id	int	PRI	no	auto_increment	
t_{start}	datetime	MUL	no		0000-00-00 00:00:00
t_{end}	datetime	MUL	no		0000-00-00 00:00:00
t_{incr}	datetime		no		0000-00-00 00:00:00
t_{trans}	datetime		no		0000-00-00 00:00:00
det_id	tinyint		yes		NULL
data_id	tinyint		yes		NULL
composite_id	int		yes		NULL
version_id	int		yes		NULL

Table 1.2: The Validation table model. The role of the validation VAL table is to quickly locate its corresponding user-defined data table in the database. It allows for the selection of the correct set of sequence identifier values Seq_id based on the user supplied metadata information (event time, detector type, data type etc.). For the keys column, PRI means primary and MUL means *multiple* which is basically an index that is neither a primary key nor a unique key i.e multiple occurrences of the same value is allowed.

share a unique IoV. In this way, the time range only appears once and just one Seq_id is reused in the data payload table.

t_{trans} : is the **transaction time** i.e the time when the data is stored in the database. Adding the transaction time in the validation table allow the virtual database to records changes to the underlying database itself and thus allow for *rollbacks*. By telling the query to ignore all data inserted into the database after a user-defined timestamp, the user can change the state of the database at any previous time. The *rollback* is relevant when one wants *freeze* a particular state of the database used during the data production process, eventhough the system continues to receive updates.

t_{incr} : is an additional **incremental time** used to resolve ambiguity on multiple existing data payload in the database for the same *context*. It can be though of an automatic version numbering scheme based on the first creation time of the table row corresponding to a parameter object. Everytime a new version of the same object is added as a row, the time is incremented by a predefined constant time τ i.e $t_{incr} \equiv t_0 + \tau$. If multiple Seq_id rows satisfy the request (span the events time, match the detector type etc.) then the system simply chooses the latest incremented time t_{incr} under the assumption that newest data input corresponds to the *best* data version (e.g. last updated calibration constants).

composite_id : denotes the **data composition identifier**. When the flag is used, it allows a logical collection of object to be splitted in more manageable smaller chunks. This allow the system to do incremental update on composite objects relevant for example in the case of partial recalibration for a chosen subset of segments in the detector planes.

such a scheme, if one has data valid within $[t_1, t_\infty[$ and need to patch within $[t_2, t_3]$, one needs first to put in the patch data within $[t_2, t_\infty[$ and then duplicate original t_1 data in $[t_3, t_\infty[$. this mechanisms turns out to be inefficient and not natural since the patch data is only valid within $[t_2, t_3[$.

`det_id`, `data_id` : are additional **detector and data types identifiers**. They are bit-wise tested against the user supplied version, detector and data types suitably converted into a single bit i.e if for `det_1` the id is 1, then the first bit is set, if for `det_1` the id is 2 then the second bit is set etc. Compact `det_id`, `data_id` allows a row in the validation table to be valid for more than one detector type (same for the data type).

The main purpose of the validation management is to provide the best possible information giving a particular user-defined *context*. Nevertheless, it should also be possible to retrieve data using more complex queries. For this reason, the system can add extension to the standard context-based query. For instance, the query can be extended to all ranges between t_{start} and t_{end} . The result of such extended query will be represented by a collection of rows which will not represent the state of the experiment at a precise time and will need additional filtering of the data. It is then the user responsibility to interpret the final results set in a meaningful way.

1.9 Persistency scheme

As already described above, the virtual database can break down a FairRoot based parameter object derived from the generic `FairParSet` class into a relational data model (two dimensional table consisting of rows and column). During the object to table mapping process the virtual database is storing also metadata information via a user-defined *context* object which information will also be stored in a relational table (validation table) together with the data payload table (primary table).

Further user request will never access the parameter data directly in the database but will query first the auxiliary validation table to get the corresponding sequence id number in order to quickly lookup the row corresponding to the parameter data object in the corresponding data table.

The overall object-to-table mapping is the responsibility of the virtual database. It can be thought of as a rudimentary SQL-based persistency mechanism ¹⁹ which is illustrated in Figure 1.12.

1.9.1 SQL processing

For all SQL-based operations, a transaction will be processed through the virtual interface using predefined SQL statement or queries. This approach has the advantage to allow a better and more centralised control of the database server load.

For example, when multiple object instances need to be stored, a SQL processor stack will buffer all the corresponding SQL statements. When an `execute()` trigger is sent by the system or simply when the processor stack reaches its limit size, the aggregated SQL commandos are executed as a bunch at once.

Another example is the SQL parsing and translation mechanism. The virtual database use as a default the MySQL language for prototyping SQL commands. Since the virtual database supports different database backends it needs a special pre-processing of the input SQL statement which is the responsibility of the central SQL statement processor class `FairDbSQLStatement`. The SQL pre-processing works as follow

¹⁹ The virtual database object persistency scheme can not preserve the exact structure of the object in memory. Instead it will simplify it. For instance inheritance and complex aggregation can not be fully preserved. Object composition can only be handled via polymorphic IO streamer functions.

- A MySQL Statement is registered in the SQL statement buffer of the processor class FairDbSQLStatement
- The backend database identifier is use by the processor class FairDbSQLStatement to trigger or not a SQL translation.
- If a translation is needed, a fast parsing is processed and a new adapted SQL statement is dynamically created with the translated tokens.
- The new SQL statement execution is then forwarded to the TSQLServer class which in turn uses the processing functions TSQLStatement::Statement(TString&) and TSQLStatement::Process()

1.9.2 Disk cache

Additional to the persistency scheme itself the virtual database is designed to minimize the I/O by using dynamic disk caching whenever possible. Fundamentally, caching realizes a performance increase for transfers of data that is being repeatedly transferred. The virtual database implements its own read-oriented caching mechanism. When a data table is being fetched from its residing location in any database, the data can on demand be copied to disk in order that by subsequent reads the same data will be fetched from the cache's (faster) intermediate storage rather than the data's residing location.

This is particularly relevant whenever some requests pull in large amount of data. One

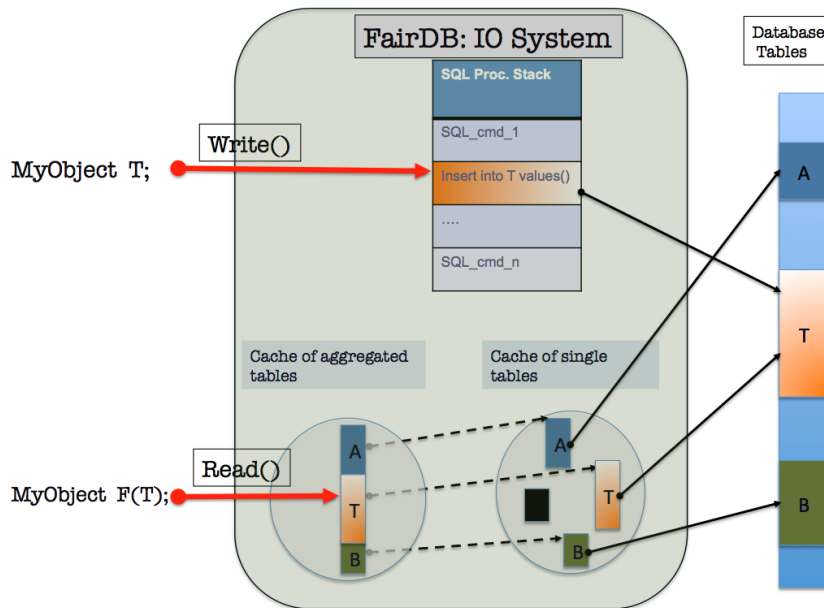


Figure 1.12: IO persistency scheme. All IO related operations are centralised and processed through the virtual interface using predefined SQL statements. Each SQL statements for Write() operations is added to a dedicated SQL processor stack which aggregates the statements before processing a real transaction. A two-levels caching mechanism is foreseen for immediate performance increase in case of repeated Read() operations.

should then avoid to load everything at once at the start of a process and then use it

repeatedly. Furthermore, the large data loaded may not be valid for all the data the user will process and reinitialisation may be again needed. Additionally more users may want access to the same parameter data and it would be not efficient for each of them to have their own copy.

To cope with these problems, the virtual database handles the actual user I/O actions via a proxy layer. In the reading access (Figure 1.12) when the user wants to access a data table a table-specific light pointer object is defined. This pointer object is very thin and suitable to be stack based in-memory and passed by value²⁰. To define the object pointer, a dedicated data request is passed through the virtual interface and the results table, which may be large, is created on the heap. The virtual interface store the big table in a predefined cache area, and the user object pointer is connected to the table. In this case to user does not own the table but only the virtual database does.

Each further data requests will first be processed in the cache and if the data is already existing in the cache it is reused. When a pointer to a table is discarded by its owner, it disconnects itself from the corresponding table it points to. Each table does an internal bookkeeping of all pointers that are connected to it. If a cached table is pointer free, it will be tagged for being dropped by the corresponding cache.

The overall mechanism can be thought of as a simple disk (level 2) caching mechanism. Caching realize immediate performance increase upon the initial read transfer of a data table when time and I/O consuming query process is repeated many times, in many jobs that process the data. As a matter of fact, for each SQL query

- SQL query has to be processed and applied to data
- all corresponding row objects have to be fetched and allocated in heap based memory
- conversion through several layer of the underlying database has to be applied to each data word of each row.

It is clear that one should avoid repeating the same time consuming query process outlined above. An obvious optimisation is to simply make an image copy of the table to disk and thus can be reloaded rapidly when required.

1.10 Connection Pooling

One of the most expensive database-related tasks is the initial creation of the connection between a client and the database server. Once the connection is established the transaction often takes places quickly.

The virtual database is designed to give the user the possibility to connect to and access data from more than one database. A connection pool is established once at initialisation time when the main SQL-based IO interface `FairDbSQLIO` is instantiated and added as a possible IO interface within the FairRoot runtime database manager class `FairRuntimeDB`.

The connection pool is responsible to maintains a pool of opened connections to different database engines, so that the application or the user can simply grab one when it needs it to execute a transaction and then hand it back, eliminating much of the long wait for the creation of connections. The connection pool initialisation is done internally using a JDBC similar URL based syntax supported by the Root SQL Server `TSQLServer` in order to connect a list of database servers.

The list order can reflect either data storage priority or data usage priority. For example, when dealing with recalibration of parameter, the probability to change the calibration

²⁰ In this case, passing data object by value will reduce possible memory leaks.

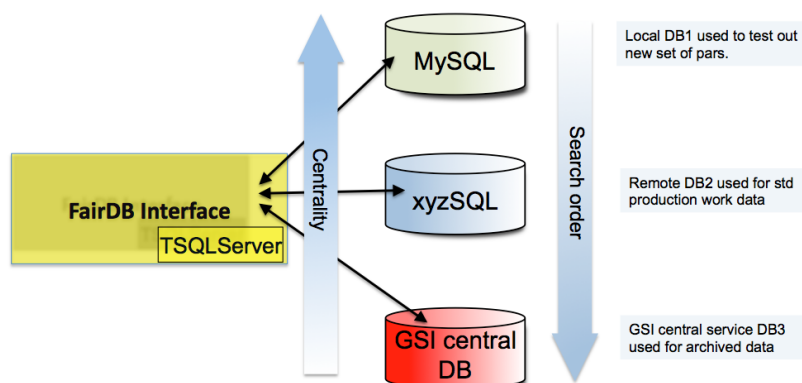


Figure 1.13: Connection pool.

constants during the day is high. The user will then take advantage to put at first position within the URL list its own local MySQL database saving the network latency time for storing or retrieving the parameter. In Figure 1.13 the system is initialised with three different connections. The user has created a local database with its own data eventually overriding parts of the official database and placed it ahead of the official GSI database. This scheme allows by construction a better load balancing on the central database service such as for example the official GSI database in Figure 1.13.

By default, the virtual database first fetch data in the first database located in the list. If the transaction fails then it tries the others in turn until the data is successfully found or all databases have been tried without success.

Part II

Database Guidelines

Handling a Database

This chapter presents how to store and access parameter data in a SQL-based database using the virtual database services of the FairRoot framework. Installation guidelines for the supported SQL databases is presented. Then the FairRoot virtual database services is detailed including how to setup multiple connections, how to setup the runtime database SQLIO manager class and how to write your own *database-aware* parameter data class.

2.1 Installing

2.1.1 Local @ GSI

If you can access the GSI Linux cluster you just need to setup the standard FairRoot environment variable `SIMPATH` in order to use predefined and compiled libraries.

The external packages distribution should be at least releases from *Sep12*, you can also use the release *Apr13* installed in the universal Cern File System (CernVMFS) namespace `/cvmfs` as follow:

```
shell> export SIMPATH=/cvmfs/fairroot.gsi.de/fairsoft/sep12
shell> export SIMPATH=/cvmfs/fairroot.gsi.de/fairsoft/apr13
```

These external packages are built using precompiled ROOT libraries containing additionally ROOT SQL drivers i.e `TSQLServer` services which are mandatory in order to use FairDb. On a standalone system one should then always install ROOT with `TSQLServer` services.

2.1.2 Standalone system

The documentation for installing and using the ROOT framework can be found at:

```
http://root.cern.ch/root/Install.html.
```

As mentioned above, one should properly install ROOT libraries including the `TSQLServer` services. On your standalone system, you can get a snapshot of ROOT and all necessary external packages needed to build and run the FairRoot using `svn checkout` for both *Sep12* and *Apr13* releases as follow:

```
shell> svn co https://subversion.gsi.de/fairroot/fairsoft/release/sep12
shell> svn co https://subversion.gsi.de/fairroot/fairsoft/release/apr13
```

In the ROOT library installation which is located at:

```
$(SIMPATH)/tools/root
```

additional options have to be added in order to enable the compilation of MySQL and PostgreSQL drivers for the `TSQLServer` interface:

```
shell> ./configure $arch
...
--enable-mysql --enable-pgsql
...
```

If for example you would like to use an Oracle database which is also supported, you need to add dedicated Oracle options in the main ROOT configure script `configure.sh`, which will in turn locate the necessary Oracle SQL-database drivers includes and libraries paths. In the next sections the guidelines to install your own SQL database will be given.

2.1.3 MySQL

MySQL is a fast, mutli-threaded, multi-user and robust SQL database server. It is completely free (GPL licenced) and mature (rather long history) product which benefits from an extensive user-base community. Furthermore MySQL is supplied as part of recent Linux distributions (Fedora/RedHat, Ubuntu etc...) and can be easily also installed from source code, rpm and other packaging systems. It is also ditributed as binary-tarball which only comes with static (.a) libraries and so, should exactly match with the system compiler libraries.

For these reasons, MySQL is definitely recommended to use as a database engine. This section provides minimal guidelines in order to install, manage and use a MySQL database. A more detailed documentation is available online at the following useful URLs.

- the MySQL main page (developer Zone):

```
http://dev.mysql.com/
```

- the MySQL documentation page (Reference Manuals):

```
http://dev.mysql.com/doc
```

- the MySQL Community Server and Cluster downloads (GPL)

```
http://dev.mysql.com/downloads/mysql
```

2.1.3.1 Installing MySQL on Linux

Ubuntu: The easiest installation of MySQL is certainly the one proposed in **Ubuntu** since MySQL (for example version 5.5 for Ubuntu 12.04) is already pre-packaged and one can just run the following command from a terminal prompt:

```
shell> sudo apt-get install mysql-server
```

Once the installation is complete, to check if MySQL server is running, type the following command:

```
shell> sudo netstat -tap | grep mysql
```

Executing the command should lead to the following results or something similar:

```
tcp        0 0      localhost:mysql    *.*      LISTEN 2556/mysqlld
```

which means that the server is active and listening on port number 2556. If the server is not running you can always issue the command:

```
shell> sudo service mysql restart
```

to restart the server at any time. Further configurations of the server can be achieved by editing the configuration file `/etc/mysql/my.cnf`. One can then change basic settings for the logging file, port number, etc.

Installation on Debian GNU/Linux: In a similar way, one can install MySQL server on a Debian GNU/Linux distribution by issuing the command:

```
shell> apt-get install mysql-client mysql-server
```

Then editing the README file located at:

```
/usr/share/doc/mysql-server/README.Debian
```

one can setup the server using the following commands:

- adding a root password from command line

```
shell> /usr/bin/mysqladmin -u root password "your-password"
```

- adding a root password editing the configure file

```
shell> cat > /root/.my.cnf
[ mysqladmin ]
user = root
password = your-password
<Control-D>
shell> chmod 600 /root/.my.cnf
```

Installation with RPMs Installing using rpm format requires root access to the machine which is hosting the database. It requires additionally the permissions to write to `/usr/local/`.

1. Download the RPMs files at:

```
http://dev.mysql.com/downloads/os-linux.html
```

2. A minimal server/client distribution requires the four following RPM files that need to be downloaded (the number 5.5.xx is a version number):

```
MySQL-5.5.xx.i386.rpm
MySQL-client-5.5.xx.i386.rpm
MySQL-devel-5.5.xx.i386.rpm
MySQL-shared-5.5.xx.i386.rpm
```

3. To install, use the rpm install commands as follow (during the installation process you should see ### marks regularly printed on the standard output):

```
shell> rpm -Uvh MySQL-5.5.xx.i386.rpm
shell> rpm -Uvh MySQL-client-5.5.xx.i386.rpm
shell> rpm -Uvh MySQL-devel-5.5.xx.i386.rpm
shell> rpm -Uvh MySQL-shared-5.5.xx.i386.rpm
```

4. Check that the packages are installed correctly using rpm commands:

```
shell> rpm -qa | grep -i mysql
```

The last rpm command will find and list all installed RPMs files that includes the string `mysql` in lower or upper case. From the listing, one can check if MySQL was previously installed and which version has been used.

If an old version is already installed in your system, you can directly upgrade the version using rpm upgrade command `rpm -Uvh` as already described above. This will upgrade an existing version or, in case there is no previous MySQL installation, install a new version.

Installation from source files Installation from source does not require root access to your machine. You first need to download the complete source tar files for MySQL at:

```
http://dev.mysql.com/downloads/os-linux.html
```

Look for the source download for MySQL version XYZ part in the page and get the corresponding tarballs. The following shows the typical steps needed to install from source:

1. unpack the tar files

```
shell> gunzip mysql-5.5.xx.tar.gz
shell> tar -xvf mysql-5.5.xx.tar
shell> cd mysql-5.5.xx
```

2. in order to compile and install in your defined (`$INSTALL_DIRECTORY`) type:

```
shell> ./configure --prefix=$(INSTALL_DIRECTORY)
                  --without-bench
                  --enable-thread-safe-client
                  --enable-shared
                  --disable-static
                  --enable-local-infile
                  ...
```

3. Then compile the packages

```
shell> gmake
shell> gmake install
shell> cd $(INSTALL_DIRECTORY)/lib
shell> ln -s mysql/libmysqlclient* .
```

4. Install the database executing the commands:

```
shell> cd $(INSTALL_DIRECTORY)
shell> ./bin/mysql_install_db
```

For the last command, it is important to issue the `mysql_install_db` from the `$(INSTALL_DIRECTORY)` (using the two steps as described above) since the direct execution using the full path `$(INSTALL_DIRECTORY)/bin/mysql_install_db` by itself may not work.

2.1.3.2 MySQL on Mac OsX 10.6 (10.7)

On Mac OsX 10.6 (Snow Leopard) or 10.7 (Lion) the installation is rather simple since it is possible to install the database on both systems using the package designed for Mac OsX 10.6.

1. Download the 64-bit Apple Disk Image (DMG) installer for Mac OS X 10.6. While the download page says that the installer is for Snow Leopard (Mac OS X 10.6), it will work fine on Lion (Mac OS X 10.7) if you follow this process.

```
http://dev.mysql.com/downloads/mysql
```

2. When the download completes, double-click on the DMG file to mount the disk image. You will see an *Opening* dialog appear. When it disappears, it will create what appears to be a new disk named `mysql-5.5.15-osx10.6-x86_64` on your desktop.
3. Double-click the new icon on your desktop. This will open the disk image in Finder and you will be able to install the database just following the installer instructions.

2.1.3.3 Compiling ROOT with MySQL driver

Once MySQL is installed, one should add in the main ROOT `configure` the following setup lines:

```
shell> ./configure linux
--enable-mysql
--with-mysql-incdir=$INSTALL_DIRECTORY/include/mysql
--with-mysql-libdir=$INSTALL_DIRECTORY/lib/mysql
```

in order to compile the ROOT MySQL client driver.

Eventually replace `linux` with the appropriate platform. If your are not using linux use

```
./configure --help
```

to get list of supported platforms and other possible options.

2.1.4 PostGreSQL

PostGreSQL comes with complete and detailed installation guides for all platforms which can be found at the PostGreSQL wiki pages

```
https://wiki.postgresql.org/wiki/Detailed\_installation\_guides
```


General User documentation, tutorials, manuals and also complete books are also available from the main PostGreSQL wiki page:

```
https://wiki.postgresql.org/wiki/Main\_Page
```

2.1.4.1 Compiling ROOT with PostGreSQL driver

Once MySQL is installed, one should add in the main ROOT configure the following setup lines:

```
shell> ./configure linux
          --enable-pgsql
          --with-pgsql-incdir=$PGSQL_INSTALL_DIRECTORY/include/
          --with-pgsql-libdir=$PGSQL_INSTALL_DIRECTORY/lib/
```

where the variable `$PGSQL_INSTALL_DIRECTORY` corresponds to the installation directory of the PostGreSQL.

If you did not explicitly specify the installation directory, the PostGreSQL will install itself to the default directory `/usr/local/include` and `/usr/include` on your system. If so, you do need to specify the `include_path` and `lib_path` in the main ROOT configure script in order to compile the ROOT MySQL client driver.

Eventually replace `linux` with the appropriate platform. If your are not using linux use

```
./configure -help
```

to get list of supported platforms and other possible options.

2.1.5 Oracle

This section supposes that an Oracle Server is available and already installed. Recently a free but limited (maximum 10 GByte storage capability) version of Oracle server, so called Express Edition, based on the Oracle Database 11_g Release 2 is available at:

```
http://www.oracle.com/technetwork/indexes/products/index.html
```

To be able to communicate with a Oracle Server with the ROOT TSQLServer services, one should first install the so called `instant-client` libraries splitted into two packages `instantclient-basic` and `instantclient-devel`.

Both are needed since only the `instantclient-evel` contains the mandatory `include-files`. `instant-client` libraries are available for the main used operating system i.e Linux, Mac OS X, Solaris SPARC, HP-UX etc.

```
http://www.oracle.com/technetwork/database/features/instant-client
```

Additionally one needs to pick up the *correct* Oracle OCCI Client to use to build. It depends on which version of the C++ compiler (generally gcc) you are using.

For instance, if ROOT is installed with gcc version 4.2.1 then you should download the corresponding OCCI tar-ball (`occi_gcc421.tar.gz`) from the Oracle OCCI download area:

```
http://www.oracle.com/technetwork/database/features/oci
```

After installing the above packages, you will need to add in the main ROOT configure script the corresponding lines:

```
shell> ./configure linux
        --enable-oracle
        --with-oracle-incdir=$(CLIENT)/instantclient_X_Y/inc
        --with-oracle-libdir=$(CLIENT)/instantclient_X_Y/lib
        ...
```

where \$(CLIENT) locates the *instant-client* installation directory.

For instance the following list shows the contents of the *instant-client* installation directory on GSILinux cluster (version: IC11_1_0_7):

```
drwxr-x-- 4 denis rz 4,0K 2012-01-09 09:24 .
drwxr-x-- 3 denis rz 23 2012-01-09 09:24 ..
-rwxr-x-- 1 denis rz 22K 2012-01-09 09:24 adrci
-rwxr-x-- 1 denis rz 29K 2012-01-09 09:24 genezi
drwxr-x-- 2 denis rz 4,0K 2012-01-09 09:24 inc
drwxr-x-- 2 denis rz 4,0K 2012-01-09 09:24 lib
-rw-r---- 1 denis rz 2,7K 2012-01-09 09:24 libaio.so
-rw-r---- 1 denis rz 2,7K 2012-01-09 09:24 libaio.so.1
-rw-r---- 1 denis rz 2,7K 2012-01-09 09:24 libaio.so.1.0.1
-r-xr-x-- 1 denis rz 36M 2012-01-09 09:24 libclntsh.so
-r-xr-x-- 1 denis rz 36M 2012-01-09 09:24 libclntsh.so.11.1
-r-xr-x-- 1 denis rz 5,7M 2012-01-09 09:24 libnnz11.so
-r-xr-x-- 1 denis rz 2,3M 2012-01-09 09:24 libocci.so
-r-xr-x-- 1 denis rz 2,3M 2012-01-09 09:24 libocci.so.11.1
-rwxr-x-- 1 denis rz 82M 2012-01-09 09:24 libociei.so
-r-xr-x-- 1 denis rz 127K 2012-01-09 09:24 libocijdbc11.so
-r-r----- 1 denis rz 1,9M 2012-01-09 09:24 ojdbc5.jar
-r-r----- 1 denis rz 1,9M 2012-01-09 09:24 ojdbc6.jar
-rw-r---- 1 denis rz 1,9K 2012-01-09 09:24 tnsnames.ora
```

After updating the ROOT configure script `configure.sh`, one can then perform the standard `gnu-make-install` steps for building the ROOT libraries with Oracle driver support. Before testing a connection, one has to add the *instant-client* library path:

```
shell> export LD_LIBRARY_PATH= /usr/lib/oracle/a_version_id/client/lib:
        ... :$(LD_LIBRARY_PATH)
```

Testing a connection to an Oracle server can be done immediately within a ROOT session using the standard `TSQLServer` service:

```
root> TSQLServer *db=TSQLServer::Connect(
                                     'oracle://any.node.com:[port] '
                                     , 'any_oracle_sid'
                                     , 'any_oracle_username'
                                     , 'any_oracle_password'
                                     );
```

In the first part of the connection string, the user should type the TCP/IP service name `any.node.com`, the `port` is the port number used for the connection which are all defined as Local Naming Parameters (`tnsnames.ora`) which can be set globally by a database administrator.

In Linux system you can check the contents of the Local Naming Parameters file at the location:

```
$(ORACLE_HOME)/admin/network/tnsnames.ora
```

For example the following shows the contents of a typical (`tnsnames.ora`) generated in the previous GSI-Linux Oracle installation ¹

```
=====
= TNSNAMES.ORA Configuration File F /oracle/ora92/NETWORK/ADMIN/tnsnames.ora
= Generated by Oracle Enterprise Manager V2
= Date.....: Thu Nov 11 15:07:57 CET 2004
=====

db.gsi.de =
(DESCRIPTION=(ADDRESS_LIST=
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac1.gsi.de)(PORT=1521))
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac2.gsi.de)(PORT=1521))
(LOAD_BALANCE=yes)(FAILOVER=yes))(CONNECT_DATA=(SERVICE_NAME=gsi4p_ha.gsi.de)
(FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC)(RETRIES=20))))

db =
(DESCRIPTION=(ADDRESS_LIST=
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac1.gsi.de)(PORT=1521))
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac2.gsi.de)(PORT=1521))
(LOAD_BALANCE=yes)(FAILOVER=yes))(CONNECT_DATA=(SERVICE_NAME=gsi4p_ha.gsi.de)
(FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC)(RETRIES=20))))

db.oracle.gsi.de =
(DESCRIPTION=(ADDRESS_LIST=
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac1.gsi.de)(PORT=1521))
(ASSRESS=(PROTOCOL=TCP)(HOST=oramac2.gsi.de)(PORT=1521))
(LOAD_BALANCE=yes)(FAILOVER=yes))(CONNECT_DATA=(SERVICE_NAME=gsi4p_ha.gsi.de)
(FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC)(RETRIES=20))))

...

```

showing the list of available connect descriptors mapped to their respective net service names. This file can also be maintained locally for use by individual clients.

In the second part of the connection string, `oracle_username` and the `oracle_password` are the name and the password needed to connect to and access objects in a database.

When the connection to a Oracle server is successful, then you can try to execute any SQL based statement to make a final test.

For example just perform a SQL query from the standard Oracle test tables (permanently written on the server for test purposes) using the same pointer to the `TSQLServer` object in memory created for the connection.

```
root> TSQLServer *res=db->Query("select foo from bar");
```

¹ Be aware that the former GSI-Oracle installation is obsolete and that Oracle as database engine will not be supported anymore by the IT department.

2.2 Handling a database server

Prior to use any client program to store or access data from a database, one should starts the database server. This is usually done by running a dedicated daemon process. Of course every database flavors has its own way to do this.

This section will only show how to handle the MySQL database server. For other database flavors the reader should refer to the corresponding documentation.

2.2.0.1 Starting the server

To start, stop or restart a MySQL server from the command line, your need first to have root access (`sudo`) and then to type the following at the shell prompt:

Linux

```
shell> /etc/init.d/mysqld start
```

```
shell> /etc/init.d/mysqld stop
```

```
shell> /etc/init.d/mysqld restart
```

Linux supporting service command

```
shell> /etc/init.d/mysqld start
```

```
shell> /etc/init.d/mysqld stop
```

```
shell> /etc/init.d/mysqld restart
```

Mac OsX

```
shell> sudo /usr/local/mysql/support-files/mysql.server start
```

```
shell> sudo /usr/local/mysql/support-files/mysql.server stop
```

```
shell> sudo /usr/local/mysql/support-files/mysql.server restart
```

To check if the server is running on your system type:

```
shell> ps -A | grep -i mysqld
```

After issuing the command, a display containing one instance of the main MySQL daemon `mysqld_safe` and one to three `mysqld` daemon processes.

For example on Linux based operating system a typical display would be:

```
shell> ps -A | grep -i mysqld
11295 pts/2 00:00:00 mysqld_safe (or safe_mysqld prior to mysql 4)
11411 pts/2 00:00:00 mysqld
11413 pts/2 00:00:00 mysqld
11414 pts/2 00:00:00 mysqld
```

and on Mac OsX:

```
shell> ps -A | grep -i mysqld
123 ? 0:00.02 /bin/sh /usr/local/mysql/bin/mysqld_safe
--datadir=/usr/local/mysql/data
--pid-file=/usr/local/mysql/data/local.pid

195 ? 74:04.12 /usr/local/mysql/bin/mysqld
--basedir=/usr/local/mysql
--datadir=/usr/local/mysql/data
--plugin-dir=/usr/local/mysql/lib/plugin
--user=mysql
--log-error=/usr/local/mysql/data/local.err
--pid-file=/usr/local/mysql/data/local.pid
```

On Mac OsX more informations about the server settings are displayed i.e where MySQL is installed `base_dir`, which data directory is used `data_dir`, the plugins directory `plugin_dir`, the error log file location `log_error` and the process ID file `pid_file`.

2.2.0.2 Server authentication

After starting a database server, it is of good practice to change the root password as soon as possible. This can be easily done with the administration tool `mysqladmin` by typing:

```
shell> mysqladmin --user=root password <root_pw>
```

where `<root_pw>` is the new password that is entered without quotes. The system then prompts for the new password, which you must enter twice. If the system won't change the root password, it is due to a known problem of the `mysqladmin` tool and you will have to use an explicit host number by typing:

```
shell> mysqladmin --user=root -host=198.134.140.95 password <root_pw>
```

where the host machine ip is substituted for `198.134.140.95`. Of course it is also possible to assign any user password with the `mysqladmin` tool. Nevertheless to create a user account, one has to do it on the server side using SQL statements from the `mysql>` prompt as explained in details in the MySQL documentation:

```
http://dev.mysql.com/downloads/mysql
```

2.2.0.3 Creating databases

Parameter data storage for an experiment requires usually the creation of several databases, each of them representing a dedicated parameter storage context i.e calibration per detec-

tor, hardware cabling (cable mapping), algorithm configuration etc. It is a good practice to create a *generic* account which is only allowed to read the data. One cannot change anything in the database using this account. For each parameter storage context, a special account which is able to make entries in the corresponding databases should be assigned to a responsible person. Only this person then can enter data using his special account and his own password. Eventually global account used to move data around in the collaboration can be created.

In the MySQL framework, a *database* corresponds to a aggregation of related tables. Each *instance* of one MySQL server can contain several *databases* and each *database* can have in turn many *tables*. Creating a new database can be done either through the `mysql>` prompt or through `mysqladmin` generic tool.

To run the MySQL client monitor just type:

```
shell> mysql -user=root -password=<root_pw>
Welcome to the MySQL monitor.  Commands end with ; or {}g.
Your MySQL connection id is 2766
Server version:  5.5.19 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates.  All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates.  Other names may be trademarks of their respective
owners.
Type 'help;' or '{}h' for help.  Type '{}c' to clear the current input
statement.

mysql>
```

From `mysql>` prompt, one can then create databases just typing for instance:

```
mysql> create database R3B_CAL
mysql> create database R3B_SIM
```

This will create two separated databases within one MySQL server instance with the respective names `R3B_CAL` and `R3B_SIM`.

2.2.1 Managing accounts

Every connection to a MySQL server requires a database account. This account is assigned to the database server and should not be confused with your normal computer account. Every MySQL database can be connected using its own set of allowed account. It is possible to manage the accounts yourself but usually this should be done by a database administrator. For practical reasons, this sections shows minimal guidelines and good practices how to manage database accounts. If you have `root` rights on the database server, you will have full control on all accounts by logging in and issuing the `GRANT` command from the `mysql>` prompt. For instance it is possible to restrict access dramatically i.e

- Read, write, delete and change privileges are fully controllable using separate commands
- Defining exactly which tables a request can access

- Defining even which columns of which table a request may access
- Setting limits on a list of IP numbers a connection is allowed to originate from.

The table 2.1 presents typical accounts that an database administrator can define for a project. Defining such standard accounts can be done by issuing the SQL GRANT statement

Accounts	Database	Passwd	Comments
reader	*, no temp.	<pw>	read access only
writer	*	<pw>	full access
reader	temp.	<pw>	access to temp. tables only

Table 2.1: Typical account types

from the `mysql>` prompt, for example as follow:

```
mysql> grant select on r3b_offline.* to reader@<HOSTNAME> identified by
"<reader_pw>";
mysql> grant ALL on r3b_temp.* to reader@<HOSTNAME>;
mysql> grant ALL on r3b_offline.* to writer@<HOSTNAME> identified by
"<writer_pw>";
mysql> grant ALL on r3b_temp.* to writer@<HOSTNAME>;
mysql> flush privileges;
```

GRANT expects privileges specifiers. These can be:

ALL giving the user basically all privileges besides grant privileges

SELECT giving the user the privilege to retrieve data from defined tables.

The access from computer is defined using the string <HOSTNAME> using the format “%.domain_name”. For instance if one can define <HOSTNAME> = %.gsi.de giving access from basically every computer within that domain name i.e ² ³ gsi.de.

2.2.2 Server Checks

Final tests of your installation is done by trying a connection with the accounts (let say <user_name>) to whom you have assigned some privileges and with the name of the machine running the server you are connecting to i.e:

```
shell> mysql -u <user_name> -h <mysql_server> -p
```

From the `mysql>` prompt, issue the following test statements:

```
mysql> use mysql;
mysql> show tables;
```

You should then be able to see among many different meta-tables or system-tables that

² Be aware that changes to the privileges tables do not take effect until the `flush privileges` command has been explicitly executed from the `mysql>` prompt. The new accounts will not be usable until you flush the privileges

³ One should avoid granting ALL on *.* as the account gets privileges you might not expect, like the ability to shutdown the server! But one can of course grant ALL on `r3b_offline.*`

the database needs, the list of the access tables which nrol who is allowed to access what parts of the database (Figure 2.1). Access depends on where people are logged in from. In

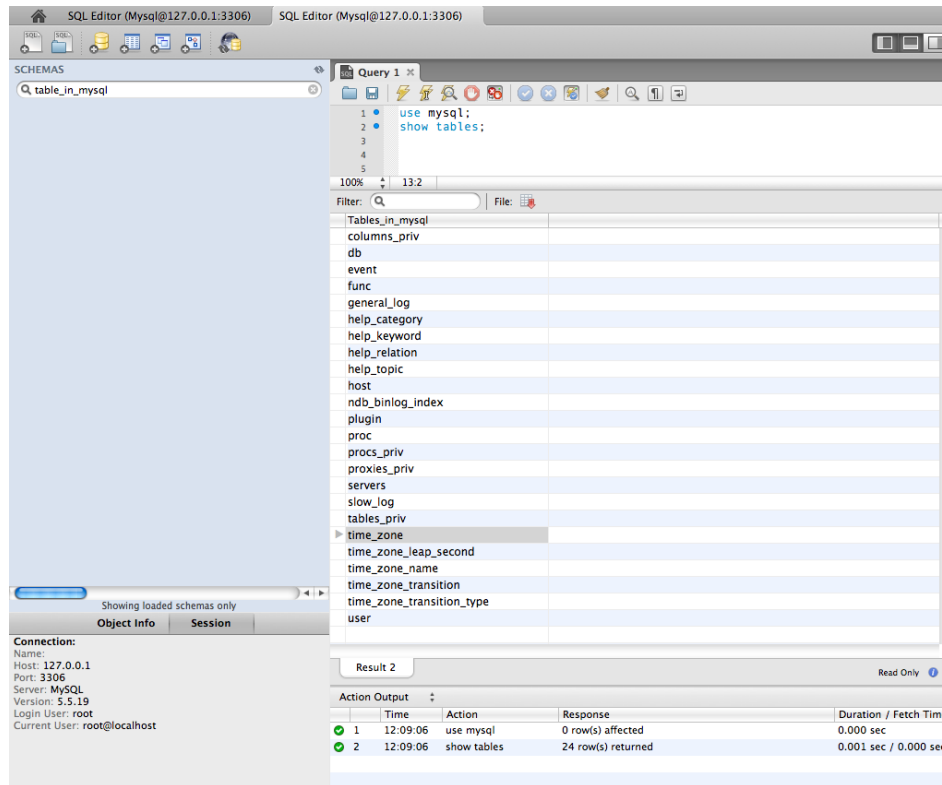


Figure 2.1: List of meta-tables shown by issuing the command `show tables;` on the main `mysql` database.

this sense it is possible to have an account which can connect from the local cluster but not from outside.

Part III
User Manual

Basic Settings

This chapter focuses on the programmatic aspects of handling condition data using the FairDB virtual database library within the FairRoot Framework. The manual presents how to setup the proper environment in order to be able to communicate with a SQL server from within the FairRoot framework. Details will be given on how to prototype a SQL-aware parameter class and how to proceed with the data version management and the SQL-persistency scheme in FairDB. For this reason, tutorials which are part of the `example` directory in the FairRoot distribution will be described.

3.1 Setting up the Environment

The user is supposed to be familiar with setting up the environment in order to use FairRoot framework for simulation and analysis. The list of relevant FairRoot environment variables are usually set by CMake generated scripts which are

```
$(VMCWORKDIR)/build/config.sh
$(VMCWORKDIR)/build/config.csh
```

where the environment variable `$(VMCWORKDIR)` points to the FairRoot main installation directory.

Additionally to be able to communicate with an external SQL server, the user has to add to the list of FairRoot environment variables additional environment variables which are needed by the FairDB virtual database library.

Using the virtual database for handling parameters modifies a standard FairRoot application into a client-server application where the server needs to be first identified. For this purpose, the virtual database needs to know the database or the list of databases that the user wants to connect to during the application and the corresponding authentication variables. These information is communicated to FairDB via a set of environmental variables beginning with `FAIRDB_TSQL_*` which defines the list of database URLs, the user name and the corresponding password.

Here is an example how to set these variables:

```
export FAIRDB_TSQL_URL="mysql://localhost/r3b;pgsql://localhost:5432/R3B"
export FAIRDB_TSQL_USER="mysql_r3b";"postgres"
export FAIRDB_TSQL_PSWD="password1";"password2"
```

Using the above setting the user communicates to FairDB that:

- two SQL database servers for parameter persistency can be used one being a MySQL server the other a PostGreSQL server. Both servers are installed on his/her local machine.

- two database users (`mysql_r3b`, `postgres`) and their corresponding passwords (`password1`, `password2`) will be used by FairDB in order to connect respectively the database servers.

Lets describe in details the differents environment variables used:

FAIRDB_TSQL_URL This variable defines a semi-colon separated list of database URLs. Each database URLs follows the ROOT `TSQLServer` semantics:

```
protocol://host[:port]/[database][?options]
where:
protocol - DBMS type , e.g. mysql etc.
host - host name or IP address of database server
port - port number
database - name of database
options - string key=value's separated by ';' or '&'
Example:
"mysql://my_defined_host:4408/r3b?Trace=Yes;TraceFile=info.log"
```

FAIRDB_TSQL_USER This variable defines the user name. The user can use a semi-colon separated list in the same manner and order as for `FAIRDB_TSQL_URL` definition if he/she requires different names for different databases in the pool. If that user name list has missing entries compared to `FAIRDB_TSQL_URL`, the first entry is used to replace the missing entries.

FAIRDB_TSQL_PSWD This variable defines eventually the user password. It can of course be considered as a security risk to leave your password in plain-text in an environment variables. One can alternatively leaves it empty. Then the user password will be prompted for at initialization time of the virtual database. As the other environment variables, it can be defined as a semi-colon separated list, the first entry defining the default if the list does not coincide with the previous ones.

These variables need to be set in order to configure not only FairRoot framework applications for read-only database access but for applications when write-access to database is required.

3.2 The SQL-IO Interface

In the introductory chapter at section 1.5.3 we have seen that within the context of the runtime database, parameter data I/O is handled in a generic way: for each type of I/O (text-based Ascii files, binary ROOT files) a dedicated set of classes is implemented, with the restriction that the main I/O interface class should be derived from the base class `FairParIo`. The `FairParIo` base class defines the basic functions to open and close any type of I/O.

Following the runtime database API, a dedicated I/O manager class `FairParTSQLIo` derived from `FairParIo` has been implemented to handle the case of SQL-based parameter I/O. The SQL-I/O interface `FairParTSQLIo` can be instantiated in a standard FairRoot macro like any other I/O interfaces and a pointer to it is stored either as first or second input in the runtime database:

```

Int_t sql_param_read(){
...
// Create a Runtime Database singleton.
FairRuntimeDb* db = FairRuntimeDb::instance();

// Create an instance of the SQL-IO interface
FairParTSQLIo* inp = new FairParTSQLIo();

// Initialize the SQL-IO interface and
// Open the connections to databases
inp->open();

// Set SQL IO interface as first input
// in the runtime database
db->setFirstInput(inp);
...
return 0;}

```

Internally the SQL-I/O Interface `FairParTSQLIo` uses the services of the pool connection class `FairDbConnectionPool` in order to be able to open multiple connection to different databases.

The same way, the other I/O Interface class implemented in the FairRoot framework uses a connection class i.e `fstream` for the text-based input files or `FairParRootFile` for the ROOT binary input files. The FairRoot I/O interfaces and their corresponding connection classes are summarized in table 3.1.

I/O class	Ascii I/O	ROOT I/O	SQL-based I/O
<code>FairParIo</code>	<code>FairParAsciiFileIo</code>	<code>FairParRootFileIo</code>	<code>FairParTSQLIo</code>
Connectors			
	<code>fstream</code>	<code>FairParRootFile</code>	<code>FairDbConnectionPool</code>

Table 3.1: The main FairRoot parameter I/O Interface classes and their corresponding connectors classes.

As for the other I/O interfaces, the SQL-I/O interface `FairParTSQLIo` holds a list of dedicated detector I/O classes `FairDetParTSQLIo` which are instantiated automatically according to the parameter container types and the input sources used by the FairRoot application.

All detector I/O classes derive from the base class `FairDetParIo` and define function to initialize and write their respective parameter containers. Additionally in the case of SQL-based I/O, all `FairDetParTSQLIo` classes are created when a connection is opened and deleted when the connection is closed.

3.3 Handling Connections

Being able to handle server connections is essential in a typical client-server application. In section 3.2 we have seen that the SQL-I/O interface `FairDetParTSQLIo` contains a pointer to a generic connector class `FairDbConnectionPool`. The connector class delivers general services to handle connections to different database servers defined within the virtual database context. In principle, access to such services is not necessary for the user, but for some applications it could be useful to have a hand on server connections. For this reason the main functions of the `FairDbConnectionPool` class is now described.

3.3.1 Multiple Connections

As shown in figure 3.1, the connector class `FairDbConnectionPool` can handle multiple connections to different database servers that have been previously set by the user with a dedicated URL list using the environment variable `FAIRDB_TSQL_URL` (cf. Section 5.4.1). It

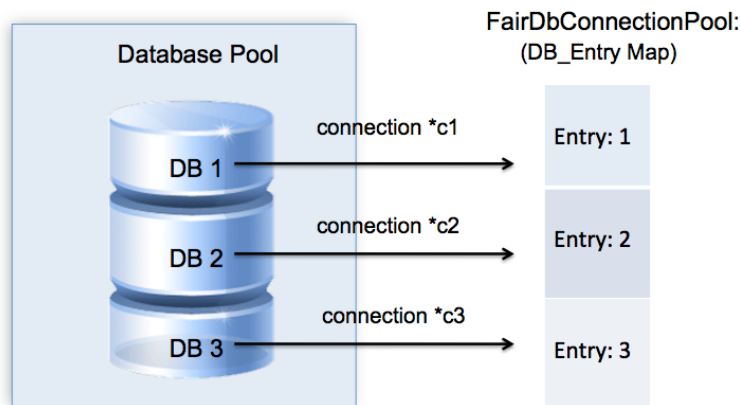


Figure 3.1: `FairDbConnectionPool` internal bookkeeping for all possible connections to databases. Every databases are store in a map with a dedicated `entry_index` corresponding to the database name.

is possible to get access to the `FairDbConnectionPool` class via the the SQL-I/O interface `FairParTSQLIo` from the macro or any compiled code:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp = new FairParTSQLIo();

// Get access to the connector class
const FairDbConnectionPool& fConnector = inp->GetConnections();
```

The `FairDbConnectionPool` class provides useful member functions in order to identify

and get information of a specific database defined in the URL list.

Let's take the example in Figure 3.1. If the user wants to get information about the database server DB1 defined at position `index=1` in the URL list, he or she can issue the following command using the `FairDbConnectionPool` connector class:

```
// Select the second database from the URL list
UInt_t db_entry = 1;

// Get Database Name giving the Database entry index
std::string db_name = fConnector.GetDbName(db_entry);

// Get Database Entry index giving a Database name
UInt_t db_entry=fConnector.GetDbNo(db_name);

// Get the corresponding URL as a string giving a Database entry index
std::string URL_def = fConnector.GetURL(db_entry)
```

3.3.2 Checking Connections

It is possible to verify the status and functionality of each opened connections using the `FairDbConnectionPool` connector class services.

The following example shows how to loop over all connections and check for each connection if the server can accept prepared SQL statements.

```
// Get the Number of Connected Databases
Int_t n_connections = fConnector.GetNumDb();
// Loop over all connections
for (Int_t iEntry = 0; iEntry < n_connections; ++iEntry) {

Bool_t trans_failed= kFALSE;

// Create a SQL Statement
auto_ptr<FairDbStatement> stmtDb(fConnector.CreateStatement(iEntry));

// Try getting a statement
if ( stmtDb.get() ) {
cout << "-Error- cannot get a statement from DB entry: " << iEntry << endl;
cout << " --> Transaction Failed ... " << endl;
trans_failed = kTRUE;
}

// Continuing if SQL check is OK
if (!trans_failed){
// do something ...
}

}
```

3.3.3 Holding Connections

In the virtual database the connections to the database close as soon as a query i.e a I/O operation is completed in order to minimize the use of resources. Indeed, in a typical analysis or simulation FairRoot application, required parameters are fetched at initialization time of the different tasks.

For such a normal use case, keeping connections to databases temporary adds just little overhead as typically there is major database reads at the beginning of the analysis application after which only little or no further database I/O will occur.

However for other cases when a lot of database I/O is expected, temporary connections are not an optimal setting. For example for applications estimating calibration parameters, the data may change and need to be stored from run to run. Another example can be if the database is linked internally to several flat files since re-opening such a database would involve re-loading of all data contained in the files.

In such cases, keeping temporary connections may quickly degrade the performance and the user should instead set all connections to be permanent. This can be done using the SQL-I/O interface `FairDetParTSQLIo` as follows:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Set all connections permanent
Int_t conn_mode=1;
inp_io->SetHoldConnection(conn_mode);
...
```

The user can also force only a single database entry to be connected permanently. This can be done from the connection pool class:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Get access to the connector class
const FairDbConnectionPool& fConnector = inp_io->GetConnections();
// Select database entry 1
Int_t db_entry = 1;
// Request database entry 1 to be permanently opened
Bool_t setConnPermanent=kTRUE;
fConnector.HoldConnectionAt(db_entry, setConnPermanent);
...
```

The user can alternatively reverse the option and make all database connections to be set temporary;

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Set all connections temporary
Int_t conn_mode=-1;
inp_io->SetHoldConnection(conn_mode);
...
```

And the same way as above when considering only a single database entry:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Get access to the connector class
const FairDbConnectionPool& fConnector = inp_io->GetConnections();
// Select database entry 1
Int_t db_entry = 1;
// Request database entry 1 to be temporary opened
Bool_t setConnPermanent=kFALSE;
fConnector.HoldConnectionAt(db_entry, setConnPermanent);
...
```

Nevertheless, the user should be cautious when using permanent connections: the overhead costs can be large when considering the number of connections that would be required on batch farms running many jobs.

3.3.4 Closing Connections

As default when closing at the end of a FairRoot application, the virtual database does not clean up parameters containers that have been eventually cached on disk since this operation can be rather time consuming. However it is possible to force the virtual database to do a complete shutdown with cache clean up by setting:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Shutdown Mode ( True, False )
inp_io->SetShutdown(kTRUE);
...
```

3.4 Caching

If a query takes a long time to complete which is always the case for large table query results, it is advantageous to store the data in and to retrieve the data from a high performance store (ideally memory or disk) either explicitly or implicitly. The process of storing the query results into a cache can increase the performance for example if the same query is done many times within the same application.

The user can enable the virtual database caching mechanism allowing certain large table query results to be written to disk from which they can be directly fetched and reloaded by subsequent applications. Such a caching mechanism can save in case of large tables as much as an order of magnitude in load time. When Caching is enabled, any query will trigger a lightweight initial check in the database to confirm that the data stored in the cache is not stale. If parameter data had changed, the cache will then be renewed with the updated parameter version fetched from the database.

To enable the virtual database caching mechanism, the user defines the directory to which he/she have read and write access. For example, the following code (the cache setting can be done in a ROOT macro) will enable caching with a cache being the current working directory.

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Caching Activation
inp_io->SetCache("./");
...
```

For some applications, cache files can be shared between users locally at a site to maximise the gain in performance. In this case, a system administrator can set up a common directory to which a the group has read/write ¹ access. The administrator should then regularly control the shared cache file size: should the cache become too large it can be removed and further repopulated by the next jobs with their current queries.

3.5 Rollback

At any time, an application process can trigger a transaction which in turn could change the state of the underlying database. This way the database changes almost constantly to reflect for example the state of the different detector during the calibration procedure.

In some cases, for example during debugging or crosschecking of calibration constants, it is mandatory to decouple application from recent updates and this operation requires database ² rollback.

3.5.1 Rollback Mechanism

In figure 3.2 a single transaction and a corresponding rollback is shown. The transaction begins when data is read from or written to the database. The initiation and termination of a transaction define a point of consistency within an application process.

A point of consistency is a time when all recoverable data that an application program accesses is consistent with other data. In figure 3.2 the state of the database is first moved from a starting point of consistency (A) to another point of consistency (B) at the end of the transaction. If during the transactional process a failure occurs or if the ending point of consistency is questionable (for example because of wrong data input) it is useful to apply

¹ Cache performance is done by doing binary I/O and consequently the cache file are platform specific. If running in a heterogeneous cluster, it is of good practice to define a platform specific directory `inp_io->SetCache($FAIRDB_CACHE/$ARCH/file_cache)`

² In database terminology, a *rollback* backs out, or cancels, the database changes that are made by the current transaction and restores changed data to the state before the transaction began.

a rollback which can be seen as the exact reverse transaction.

When a rollback operation is applied, the database will ignore all changes that occurred after the defined rollback date: it is like if the virtual database backs out the changes of the wrong transaction and restore the data consistency that existed when the unit of work was initiated.

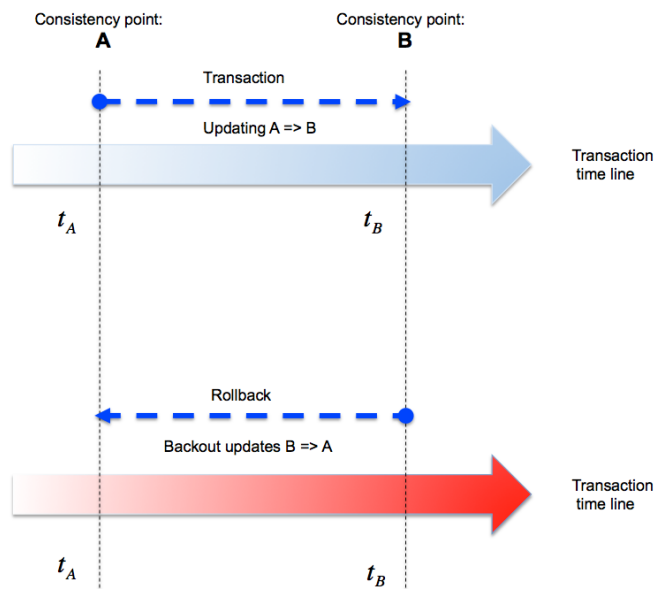


Figure 3.2: The Rollback mechanism. The top picture shows a single transaction operation. The picture below shows the rolling back changes from the corresponding transaction.

3.5.2 Rollback Configuration

Using the virtual database, it is possible to require a global rollback ³ date to let's say October 23th 2012 by:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Rollback Activation
inp_io->SetRollback("2012-10-23","*");
...
```

This forces the virtual database to ignore all parameter data inserted after this date for all subsequent queries.

³ Hours, minutes and seconds can be omitted and are defaulted to 00:00:00. The user who cares to be more precise can still define a rollback date adding hours, minutes and second `inp_io->SetRollback("2012-10-23 12:45:09","*")`.

The user can use more selective command for the rollback specifying a single table or a group of tables sharing a common prefix i.e:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Selective Rollback
inp_io->SetRollback("2012-10-01","CAL*");
inp_io->SetRollback("2012-08-14","CALTOF*");
inp_io->SetRollback("2012-07-25","CALTOFTDC");
...
```

In the above example, the single table `CALTOFTDC` have its values fixed at July 25th 2012 values, all other `CALTOF` tables have values fixed at August 14th 2012 values and all the rest of the tables will be fixed at October 1th 2012 values for the virtual database. The ordering of the rollback commands is irrelevant.

More advanced feature is to choose between ordering time dimensions for the rollback mechanism. Usually the transaction time is the proper ordering time dimension for a rollback and is used as a default by the virtual database. The user can change the time ordering on a table by table basis using the `SetRollbackMode()` function of the SQL-I/O interface `FairParTSQLIo`:

```
// Create an instance of the SQL-IO interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Selective Rollback Using
// 0 (Transaction Time)
// 1 (Incremental Time)
inp_io->SetRollbackMode(1,"CAL*");
...
```

In the above example incremental time will be use instead of the transaction time for the rollback mechanism. Incremental time is useful to notice the time of insertion in a local database. The transaction time is a global time of original creation of a parameter object and is valid for all databases. This way using incremental time will basically undo recent changes done to a local database and effect will then be connected to this specific database, while using transaction time will undo recently changed data in a universal and uniform way for all databases in the pool.

3.6 Ordered Queries

When no query results ordering is required, the database server is not forced to deliver the data in a particular order. In other words, the same application running twice while connected to the same database could query results sets that still contain the same data but with different row ordering. Since normally the row ordering is irrelevant, for the application it does not really matter.

However there are situations where identical results, even in result sets row ordering, is required. This is the case during code development and debugging phase, or simply when

checking the compatibility between different databases.

For such cases, it is possible to force the virtual database to re-order result sets using the `row_id` column which is defined for all data payload tables. The Query ordering mechanism is set through the SQL-I/O interface `FairParTSQLIo`:

```
// Create an instance of the SQL-I/O interface
FairParTSQLIo* inp_io = new FairParTSQLIo();

// Set Query Ordering
inp_io->SetQueryOrdering();
...
```

3.7 Error Handling

In the client-server application it is crucial to be able to disentangle errors coming from the server to those coming from the database interface itself.

The virtual database uses from the beginning c++ exceptions for error handling and additionally implements a global error logging mechanism. Within the virtual database context errors coming from the server, which are mainly SQL syntax errors, are handled by the `TSQLServer` and sent to the standard output. These errors always have `Error in <TSQLServer> Code: XXXX` standard ROOT prefix at the beginning of the error message containing a 4 digits error code:

```
Error in <TMySQLServer::TMySQLServer>: Code: 1049 Msg: Unknown database 'r3ba'
```

Errors coming from the virtual database itself are treated using standard C++ exceptions and are automatically logged by the central logger manager class `FairDbExceptionLog`. All virtual database internal errors or information messages are timestamped and shows the relevant classname, function name and line in the source code where precisely the error occurred.

For Instance the corresponding error caught within the virtual database corresponding to the connection problem showed above is:

```
[2013/11/25 12:21:08] -E- FairDb: FairDbConnection.cxx::Open():[143]
$ -I- FairDbConnection::Open() Failing to open: mysql://localhost/r3ba
for user scott and password tiger (attempt 1)retrying ...
```

The global log file can be defines by setting the following environment variable.

```
export FAIRDB_LOGFILE_DB=/Users/denis/errors.log
```

If the environment variable `FAIRDB_LOGFILE_DB` is empty then the default log file name will be use `fairdbinfo.log` and the file will be located in the current directory where the user executes the macro.

The user can append his/her own message to the central log by adding in the implementation code the following include:

```
#include "FairDbLogService.h"
```

and by using the following precompiler log macro:

```
DBLOG("FairDb", FairDbLog::kWarning) < "FairDbTutParBin Store() " < endl;
```

issuing a `Warning` message that will be sent to the central logger.

3.7.1 Exceptions Logging

During the processes execution, all exceptions coming from the virtual database are registered the central logger class `FairDbExceptionLog`. The exception logger class is a singleton and delivers at any time the contents of all exceptions that have been recorded during process execution.

The global exception log contents can be printed using the following code fragment:

```
DBLOG("MyLib",FairDbLog::kInfo) <<"Central Exception Log Contents"
<< FairDbExceptionLog::GetGlobalErrorLog();
```

The user can also access potential errors that occurred when executing a query to the database. The reader template `FairDbReader` is linked to a dedicated exception logger class that holds all exceptions (if any occurred) recorded when the user query was executed. The user can check and print the exception log contents associated to a query as follows:

```
FairDbReader<FairDbTutPar> MyReader;
const FairDbExceptionLog& eLog(MyReader.GetResult()->GetErrorLog());
if(eLog.Size()==0) DBLOG("MyLib",FairDbLog::kInfo) <<"No errors found"<<endl;"
else DBLOG("MyLib",FairDbLog::kInfo) <<"Errors found:"<<eLog<<endl;"
```

3.7.2 Output Log File

Different level of verbosity for the logging mechanism can be set from the SQL I/O interface `FairDbParTSQLIo`.

```
FairParTSQLIo* inp2 = new FairParTSQLIo();
// Set logging Verbosity level
// Level Active Streams
//
// 0 Warning+Errors(logged),
// 1 Info (logged),
// 2 Debug+Info (logged)
// 3 Debug+Info (logged+standard output)

inp2->SetVerbosity(1);
```

Level 0 verbosity is set by default (only errors and warnings are logged). When Level 1 verbosity is chosen, all information collected during the running process are registered and written in the central log file. During the process execution, all information or error messages are appended to the central log file with the associated timestamp (date and time) of the message occurrence. Additionally the class name, the class method name and the exact line in the code from where the message was sent is shown. The header of the log

file gives a full diagnosis of the test connections with the different servers present in the priority list.

```
[2013/11/28 09:02:14] -I- FairDb: FairDbTableInterfaceStore.cxx::SetLoggingStreams():[439] $ FairDb logging service: opened.
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::FairDbConnection():[37] $ Creating a DB connection
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Open():[152] $ Successfully opened connection to: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::FairDbConnection():[40] $ successfully opened connection to: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::FairDbConnection():[78] $ this client, and MySQL server (MySQL 5.5.19) does support prepared statements.
[2013/11/28 09:02:14] -I- FairDb: FairDbStatement.cxx::ExecuteQuery():[63] $ Server:r3b SQL:Select * from FAIRDBGLOBALSEQNO where 1=0
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Close():[192] $ closed connection: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbConnectionPool.cxx::FairDbConnectionPool():[112] $

FairDbConnectionPool:: Server Status :
Status URL: Closed (auth) mysql://localhost/r3b

[2013/11/28 09:02:14] -I- FairDb: FairDbTableInterfaceStore.cxx::SetLoggingStreams():[439] $ FairDb logging service: opened.
[2013/11/28 09:02:14] -I- FairDb: FairDbStatement.cxx::ExecuteQuery():[63] $ Server:r3b SQL:select * from FAIRDBGLOBALSEQNO;
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Open():[152] $ Successfully opened connection to: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Close():[192] $ closed connection: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbTableInterfaceStore.cxx::GetTableInterface():[229] $ create a FairDbTableInterface 0x7fff5f800000
FairDbTableInterfaceStore::FairDbTableInterfaceStore()
[2013/11/28 09:02:14] -I- FairDb: FairDbTableMetaData.cxx::FairDbTableMetaData():[28] $ create for table # FAIRDBTUTPAR
[2013/11/28 09:02:14] -I- FairDb: FairDbTableMetaData.cxx::FairDbTableMetaData():[28] $ create for table # FAIRDBTUTPARVAL
[2013/11/28 09:02:14] -I- FairDb: FairDbProxy.cxx::CreateMetaData():[531] $ Create meta-data for table: FAIRDBTUTPAR
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Open():[152] $ Successfully opened connection to: mysql://localhost/r3b
--- fairdbinfo.log Top L12 (Fundamental)---
```

Figure 3.3: Log File Header (`fairdbinfo.log`) when verbosity level 1 (info) is set. The virtual database first connect to the server and gives a full diagnosis of the connection.

```
[2013/11/28 09:02:14] -I- FairDb: FairDbTableInterfaceStore.cxx::Query():[85] $ Query fulltimewindow 1
[2013/11/28 09:02:14] -I- FairDb: FairDbCache.cxx::Search():[165] $ Primary cache search of table: FAIRDBTUTPAR for: [ Gfil Data
[2013-10-22 15:00:00.000000000Z] with Version: 0
[2013/11/28 09:02:14] -W- FairDb: FairDbCache.cxx::Search():[170] $ Primary cache search failed :: (derived-cache-1) is empty
[2013/11/28 09:02:14] -I- FairDb: FairDb.cxx::SetTimeWindow():[130] $ Set time window: 864000 for: FAIRDBTUTPAR
[2013/11/28 09:02:14] -I- FairDb: FairDb.cxx::GetTimeWindow():[34] $ Return time window 864000 for FAIRDBTUTPAR
[2013/11/28 09:02:14] -I- FairDb: FairDb.cxx::GetTimeWindow():[34] $ Return time window 864000 for FAIRDBTUTPAR
[2013/11/28 09:02:14] -I- FairDb: FairDbProxy.cxx::QueryValidity():[293] $ db_id: 0 SQL query: select * from FAIRDBTUTPARVAL where
TimeStart <= '2013-11-01 15:00:00' and TimeEnd > '2013-10-12 15:00:00' and DETID = 8 and DATAID = 1 and Version = 0 order by
TIMEINCR desc;
[2013/11/28 09:02:14] -I- FairDb: FairDbStatement.cxx::ExecuteQuery():[63] $ Server:r3b SQL:select * from FAIRDBTUTPARVAL where
TimeStart <= '2013-11-01 15:00:00' and TimeEnd > '2013-10-12 15:00:00' and DETID = 8 and DATAID = 1 and Version = 0 order by TI
MEINCR desc;
[2013/11/28 09:02:14] -I- FairDb: FairDbConnection.cxx::Open():[152] $ Successfully opened connection to: mysql://localhost/r3b
[2013/11/28 09:02:14] -I- FairDb: FairDbValRecord.cxx::Fill():[159] $ FairDbValRecordord for row: 0: Idet_id 0x00081data_id 0x00011
2013-10-30 13:22:33.000000000Z
2038-01-19 03:14:07.000000000Z
origin: Added Conditions: seq_id: 900000111 combo_id: -1 version_id: 0
[2013/11/28 09:02:14] -I- FairDb: FairDbValRecord.cxx::Fill():[159] $ FairDbValRecordord for row: 1: Idet_id 0x00081data_id 0x00011
2013-10-28 11:12:07.000000000Z
2038-01-19 03:14:07.000000000Z
origin: Added Conditions: seq_id: 900000109 combo_id: -1 version_id: 0
[2013/11/28 09:02:14] -I- FairDb: FairDbValRecord.cxx::Fill():[159] $ FairDbValRecordord for row: 2: Idet_id 0x00081data_id 0x00011
2013-10-28 11:00:33.000000000Z
--- fairdbinfo.log 13% L59 (Fundamental)---
```

Figure 3.4: Snapshot of the written log file (`fairdbinfo.log`) when verbosity level 1 (info) is set. Crucial in the debugging phase: all executed SQL statements are visible (for example a SQL time ordering algorithm is underlined in yellow) and can be investigated from the process central log file.

As shown in Figure 3.3, the log file header tells :

- If the test connection to server was successful
- Which server (name and version) is then connected
- If the client and the server are supporting prepared SQL statements

- Which URL is used to connect the server

If for any reason the test connection fails, the virtual database will try repeatedly to connect to the non-responding server. If the number of connection trials exceeds 10, the server is marked as non responding in the priority list and will not be further used.

Figure 3.4 shows the typical collection of messages coming from the virtual database during execution. The logged information includes also the details of executed SQL algorithms.

Because all messages written in the central log file are timestamped, it eases the debugging of a potential data corruption in the database. Indeed the user can always relate the error timestamp written in the log with the timestamp stored in the auxiliary validity table in the database to locate a potential data corruption.

C++ Parameter Object to Table Mapping

This chapter describes how to implement the C++ parameter object to relational table mapping. It describes in details which SQL related virtual methods of the parameter container class need to be overloaded and how to implement them in order to perform the intrinsic SQL I/O.

4.1 Creating SQL-aware Parameter Container

In order to create a SQL-aware Parameter container class, the user needs the following functionalities

- An object data member to relational database table mapping. This mechanism involves the automatic creation of a database table and the corresponding streaming functions.
- Versioning management by storing and retrieving data to and from the database.

These additional functionalities have been added to the FairRoot basic parameter class `FairParSet` by changing the inheritance dependencies from the top. As shown in Figure 4.1 `FairParSet` inherits from the main object to table mapper class of the virtual database library `FairDbObjTableMap`.

Redesigning that way the FairRoot parameter class will add to all user parameter classes the additional SQL I/O functionality without disrupting the former Ascii and Root file I/O services of the runtime database. Nevertheless the redesign of the parameter class creates inherently a backward compatibility issue. Since the inheritance of the parameter class has been changed, their signature in a ROOT binary formatted file will be different and it is not possible to retrieve parameter object which were stored using older versions of FairRoot library ⁴. This section focuses on the object to relational database table I/O and more precisely how to prototype the C++/SQL mapping in the parameter container class itself. Later on the terminology *intrinsic SQL I/O* will be used to distinguish between the part in the SQL I/O mechanism corresponding to the C++/SQL mapping and the overall SQL I/O mechanism i.e the retrieval and storage of condition data from/to the database system using versioning management. The overall SQL I/O will be detailed in the next chapter.

4.1.1 FairRoot Tutorials

To illustrate in a simple manner how to create SQL-aware Parameter Container within FairRoot using these new functionalities inherited from the virtual database we focus on

⁴ In FairRoot library versions older than October 2013, the parameter base class `FairParSet` inherited from the ROOT class `TNamed`. If a `FairParSet` object from older FairRoot library version was stored in Root Binary format, the generated binary stream will not be compatible with FairRoot versions newer than October 2013.

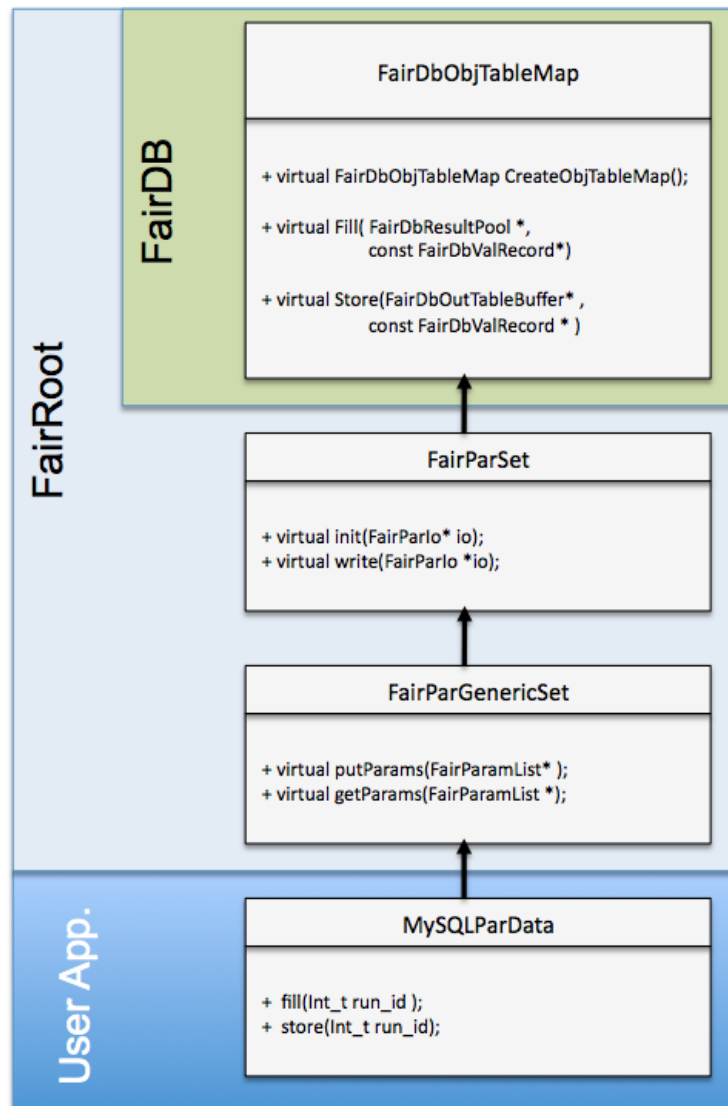


Figure 4.1: FairRoot Parameter Class inheritance. A user parameter container class inherits from both runtime database and virtual database I/O functionality simply by subclassing the FairRoot generic parameter class `FairParGenericSet`.

the examples available in the FairRoot distribution from October 2013 and located within the `example` directory, more precisely within the `Tutorial5` subdirectory.

Be aware that the `example` directory will not be fetched by default to you working FairRoot directory and you can check it out from the `fairbase` repository issuing the following command from you working directory:

```
svn co https://subversion.gsi.de/fairroot/fairbase/trunk/example/ example
```

You have to modify the main `CMakeLists.txt` in the top FairRoot directory in order to add the example in the compilation process between Lines 130 and 150.

```
add_subdirectory (fairtools)
add_subdirectory (base)
add_subdirectory (dbase)
add_subdirectory (geobase)
add_subdirectory (parbase)
...
add_subdirectory (example)
...
```

Once this is done you will have the full example sub directory containing many tutorials: The `Tutorial5` directory is divided into 2 subdirectories `src` and `macros` containing the

```
drwxr-xr-x 14 denis staff 476B Oct 2 15:59 ./
drwxr-xr-x 52 denis staff 1.7K Oct 21 14:29 ../
drwxr-xr-x 8 denis staff 272B Oct 2 09:49 .svn/
-rw-r--r-- 1 denis staff 235B Oct 2 15:59 CMakeLists.txt
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 Tutorial1/
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 Tutorial2/
drwxr-xr-x 11 denis staff 374B Oct 2 15:44 Tutorial3/
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 Tutorial4/
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 Tutorial5/
drwxr-xr-x 17 denis staff 578B Oct 2 09:49 gconfig/
drwxr-xr-x 12 denis staff 408B Oct 2 09:49 geometry/
drwxr-xr-x 10 denis staff 340B Oct 2 09:49 mcstack/
drwxr-xr-x 25 denis staff 850B Oct 2 09:49 passive/
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 rutherford/
```

example source code and the steering ROOT macros respectively. The simple parameter

```
drwxr-xr-x 13 denis staff 442B Nov 7 10:59 ./
drwxr-xr-x 6 denis staff 204B Oct 2 09:49 ../
drwxr-xr-x 8 denis staff 272B Oct 24 16:03 .svn/
-rw-r--r-- 1 denis staff 711B Oct 15 12:16 CMakeLists.txt
-rw-r--r-- 1 denis staff 1.3K Oct 2 09:49 FairDbTutAccessRtdbTask.cxx
-rw-r--r-- 1 denis staff 1.0K Oct 2 09:49 FairDbTutAccessRtdbTask.h
-rw-r--r-- 1 denis staff 3.1K Oct 23 14:44 FairDbTutContFact.cxx
-rw-r--r-- 1 denis staff 534B Oct 24 15:41 FairDbTutContFact.h
-rw-r--r-- 1 denis staff 508B Oct 15 12:16 FairDbTutLinkDef.h
-rw-r--r-- 1 denis staff 6.7K Oct 24 15:40 FairDbTutPar.cxx
-rw-r--r-- 1 denis staff 3.7K Oct 24 13:21 FairDbTutPar.h
-rw-r--r-- 1 denis staff 8.4K Oct 24 16:02 FairDbTutParBin.cxx
-rw-r--r-- 1 denis staff 4.1K Oct 24 16:02 FairDbTutParBin.h
```

container class `FairDbTutPar` used to show the basic functionalities of the runtime database in the `Tutorial1` has been adapted to be SQL-aware. We will describe in details the needed

modifications on `FairDbTutPar`.

Within the runtime database context (Figure 4.1), any parameter container class can in principle inherit from either

- `FairParSet` : the main base parameter class. Using this inheritance level gives the user the possibility to define his/her own detector I/O interfaces in order to store or retrieve parameter from Ascii or ROOT files using the functions `FairParSet::init(FairParIo *)` and `FairParGenericSet::write(FairParIo *)`.
- `FairParGenericSet`: next inheritance level for the parameter class allowing the user to have automatized I/O functionality using the functions `FairParGenericSet::getParams(FairParamList *)` and `FairParGenericSet::putParams(FairParamList *)`

Within the virtual database context, inheriting from either `FairParSet` or `FairParGenericSet` is equivalent since both classes inherits from the main object to table mapper class. In both cases, SQL I/O virtual functions are available and the user just needs to implement these functions in order to store or retrieve the parameter object from a database.

In the `Tutorial5` examples, the implemented parameter container classes inherits from `FairParGenericSet` which is chosen for simplicity.

4.1.2 Parameter Class Ownership

For its internal memory management mechanism, the virtual database needs to own any SQL-aware parameter class. Indeed, since a caching mechanism could be involved by multiple reads of the same parameter container, only the virtual database can decide if the parameter object can be destroyed.

Unfortunately the runtime database manager class `FairRuntimeDb` owns all container parameter by default. To avoid possible conflict between the virtual database and the runtime database internal memory management, the standard parameter class constructor has been modified by adding a boolean flag `Bool_t own` which has the following meaning

- `Bool_t own=kTRUE` The parameter object ownership is given to the database interface. This option is **mandatory** if parameter object caching is required.
- `Bool_t own=kFALSE` The parameter object ownership is given to the runtime database. This option should be used if the parameter object standard I/O to Ascii and ROOT file is done via the runtime database. It should not be used in case of an SQL-aware parameter container since it will inevitably lead to multiple deletion problems.

The following code shows how the parameter container class `FairDbTutPar` constructor is defined. Beside of the standard constructor arguments (`name`, `name`, `contex`), one set the boolean flag `Bool_t own = kTRUE` as default in order to give the ownership to the database interface.

```

class FairDbTutPar : public FairParGenericSet
{
// Default constructor
FairDbTutPar();

// Constructor gives ownership back to the virtual database
FairDbTutPar (const char* name="FairDbTutPar",
const char* title="Tutorial parameter",
const char* context="TestDefaultContext",
Bool_t own=kTRUE);
...
}

```

Additionally the virtual database requires that the parameter class provides a public default constructor in order to keep a object copy of every type of parameter container class e.g.:

```

FairDbTutPar()::FairDbTutPar()

```

4.2 Creating the Object Table

Storing a C++ parameter container object in a database supposes that the corresponding database table already exists or can automatically be generated from the C++ class itself. The exact mapping between the C++ class persistent data members and the corresponding relational database table can be done by implementing the function `GetTableDefinition()` defined as virtual function in the object table mapper class `FairDBObjTableMap`.

```

class FairDbTutPar : public FairParGenericSet
{
...
// SQL table description for the parameter class
virtual std::string GetTableDefinition(const char* table_name = 0);
...
}

```

The argument of the `GetTableDefinition(const char* table_name = NULL)` is the table name used to create the table in the underlying database system. A simple and good practice for choosing the table name is that a table name should be the name of the parameter class object upper-cased. In our example, the table name chosen is `FAIRDBTUTPAR` which corresponds to the parameter class object named `FairDbTutPar`.

4.2.1 Naming a Table

Since the virtual database supports different database engines, additional table naming convention rules should be observed. In order to remain compatible not only to MySQL but also to PostgreSQL and ORACLE:

- **Always use upper case letters.** Some database server (like MySQL) are generally case sensitive i.e it is possible to have a table `MYCALIB` and a separate table `Mycalib`.

Unfortunately other database engines, like ORACLE do not distinguish between upper and lower cases, so that table MYCALIB is the same as table MyCaLib !

- **Limit the length of table name** to a reasonable 30 characters long string
- **Avoid any common name** that could enter in conflict with reserved SQL words within the database engine. For example, never use VIEW or MODE if you are using ORACLE. Another example, do not use OFFSET as it is a reserved word in PostGreSQL. Many other reserved words exist that the user should be aware of. Unfortunately it is very common to develop an application and schema, use it for years on a specific database system and then, when porting to another databse, discover that a table name or even a column name is a reserved word on the target database server. It is important to check for these reserved words during your initial application development. Following are links to reserved words in the differents database flavours supported in the virtual database:

[MySQL v5.7 Reserved Words](#)

[PostGresSQL v9.3 Reserved Words](#)

[Oracle v10g Reserved Words](#)

All these naming conventions and restrictions apply not only to table names but also to column names.

4.2.2 Describing a Table

We will use as an example the parameter class `FairDbTutPar` from the `$(VMCWORKDIR)/example/Tutorial5` directory. A closer look to this class shows that the data member are:

```
class FairDbTutPar : public FairParGenericSet
{
...
/**
Persistent Data Members Section
*/

// (Detector Top-Strip Data Parameters)

// Strip pitch on top wafer side
Double_t fTopPitch;
// Anchor point of top strip 0
Double_t fTopAnchor;
// Number of Frontend attached to top wafer side
Int_t fTopNrFE;
// Frontend type name
std::string fFeType;
...
}
```

As already mentioned, in order to create the corresponding table the user simply overloads the function `GetTableDefinition(const char* table_name = NULL)`. The function basically generates the SQL statement (the default uses MySQL syntax) to create the table as:

```
string FairDbTutPar::GetTableDefinition(const char* Name)
{
string sql("create table ");
if ( Name ) { sql += Name; }
else { sql += "FAIRDBTUTPAR"; }
sql += "( SEQNO INT NOT NULL,";
sql += " ROW_ID INT NOT NULL,";
sql += " TOPPITCH DOUBLE,";
sql += " TOPANCHOR DOUBLE,";
sql += " TOPNRFE INT,";
sql += " FETYPE TEXT,";
sql += " primary key(SEQNO,ROW_ID))";
return sql;
}
```

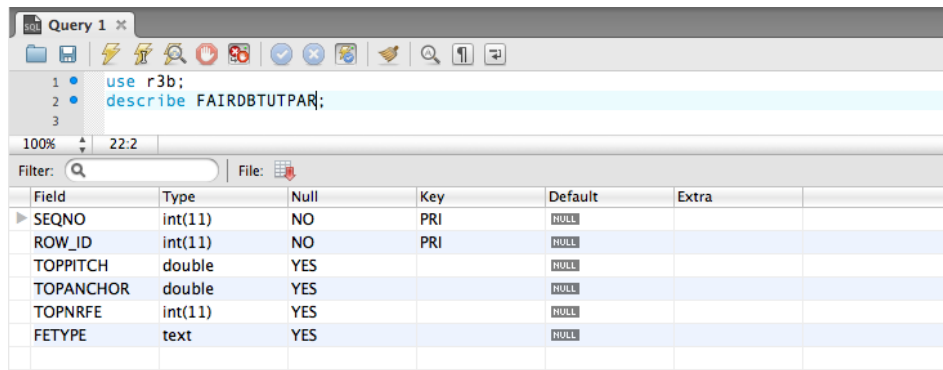
The function returns then the SQL statement as a C++ `string` which will be forwarded to the virtual database and finally executed internally using the ROOT `TSQLServer` services as the following proper SQL statement:

```
CREATE TABLE FAIRDBTUTPAR (
    SEQNO INT NOT NULL,
    ROW_ID INT NOT NULL,
    TOPPITCH DOUBLE,
    TOPANCHOR DOUBLE,
    TOPNRFE INT,
    FETYPE TEXT,
    PRIMARY KEY(SEQNO,ROW_ID))
```

Executing the SQL statement creates the table with one to one correspondence between defined columns name with corresponding field types and the C++ parameter class data members name with corresponding data types. The figure 4.2 shows how the table is created and described within the database system (MySQL example). Only two additional columns i.e `SEQNO` and `ROW_ID`⁵ with the same corresponding field type `INT NOT NULL` need to be added to the table in order to fulfill the internal virtual database version management table design. Both attributes in combination are declared as unique primary key using the SQL statement `PRIMARY KEY(SEQNO,ROW_ID)`. In the virtual database context table rows uniqueness is achieved by ensuring that for a given `SEQNO` each row has a different value of `ROW_ID`.

The user will have to add systematically these additional SQL statements associated with `SEQNO ROW_ID` to for every table created since the virtual database uses internally the

⁵ The `SEQNO` and `ROW_ID` are additional column used by the virtual interface as meta-data column to uniquely identify the table. Their values are automatically generated and so are not part the parameter data itself. Their combination ensures that every row in the user defined table is unique which is a standard practice in database design.



Field	Type	Null	Key	Default	Extra
SEQNO	int(11)	NO	PRI	NULL	
ROW_ID	int(11)	NO	PRI	NULL	
TOPPITCH	double	YES		NULL	
TOPANCHOR	double	YES		NULL	
TOPNRFE	int(11)	YES		NULL	
FETYPE	text	YES		NULL	

Figure 4.2: FairDbTutPar corresponding relational table as it is created in MySQL database. The table creation is done internally by the virtual interface using the user definition implemented with the `FairDbTutPar::GetTableDefinition(const char* table_name)` virtual function.

SEQNO and ROW_ID attributes to uniquely identify each records in the table.

4.2.3 Transient Table

During execution the user can also create transient tables which have the following characteristics:

- The table lifetime is linked to the FairRoot process lifetime i.e transient tables are automatically deleted at the end of the user's process.
- If the transient table has the same name of a persistent table within the same database, the transient table data will overrides the persistent data table.
- The transient tables are strictly linked to the process that actually has created them meaning that any user running the same process will not see the transient table.

Transient tables are useful in testing new structure for an existing table i.e

- Extending the number of the table columns
- Changing the data types

To create a transient table, the user needs to call the method:

```
Int_t
FairDbConnectionPool::CreateTransientTable(
    string & table_name,
    string & table_description
);
```

The following example code shows how to define a transient table for the FairDbTutPar parameter container:

```

// Create an instance of the SQL-I/O interface
FairParTSQLIo* inp = new FairParTSQLIo();

// Get access to the connector class
const FairDbConnectionPool& fConnector = inp->GetConnections();

// Define the table with a descriptor string
string table_descr = "( SEQNO INT,
                      TOPPITCH DOUBLE,
                      TOPANCHOR DOUBLE,
                      TOPNRFE INT,
                      FETYPE TEXT)";
// Create the FairDbTutPar table as transient
Int_t dbEntry = fConnector.CreateTransientTable("FairDbTutPar",table_descr);

// check for errors
if ( dbEntry < 0 ) {
cout << "Cannot use database for transient tables" << endl;
}

```

Basically, the user gives as arguments :

- The table name
- The description of data field and type as a C++ `std::string`. The description should be with parenthesis and comma separated following the MySQL CREATE TABLE command syntax.

It is a good practice when using transient table to setup the priority list in a way that the first database of the list acts as a temporary database, called for example `temp`, which provides for every user write-access to all tables since write-access is also needed to create transient table.

For instance, the user can set the environment variable as follows:

```

export FAIRDB_TSQL_URL= " mysql://localhost/temp;
                        mysql://localhost/r3b;
                        pgsq1://localhost:5432/R3B"

```

4.3 Parameter Intrinsic SQL I/O

The main goal of the virtual database is to deliver a set of SQL-free abstract interfaces that isolate the user code from the relational implementation i.e users write the same code for all database backends. This functionality is particularly important when SQL I/O and more precisely SQL to C++ types mapping is concerned since there is no standard ANSI SQL data types defined that could eventually help this process.

4.3.1 C++/SQL Data Types Mapping

The C++/SQL data type mapping process distinguishes between single and a user-defined type data type. For single data type a direct C++ to SQL data type is possible. Any user-defined data types cannot be stored directly into a table but will have to be converted to compact binary string (TEXT) or binary stream (BLOB) SQL types. For instance this conversion is required for :

- Dynamic or static arrays of single data ⁶ type
- C++ structures, unions or enumerations
- Any types based on other existing types: pointer to a user-defined class or data types created using the keyword `typedef` etc...

4.3.2 Data Representation

There is a strict mapping between database column types and parameter container class data members, although in cases of user-defined data types, where a conversion to formatted binary is required, one column type can be used to load more than one type of parameter class data member.

C++ Types	MySQL	PostgreSQL	ORACLE	Capacity
Short_t	TINYINT	INTEGER	NUMBER(4)	8 bit
Short_t	SMALLINT	SMALLINT	NUMBER(6)	16 bit
Int_t	TINYINT	SMALLINT	NUMBER(4)	8 bit
Int_t	SMALLINT	SMALLINT	NUMBER(6)	16 bit
Int_t	INT	INTEGER	NUMBER(11)	32 bit
Float_t	FLOAT	REAL	FLOAT(32)	32 bit
Double_t	DOUBLE	DOUBLE PRECISION	FLOAT(64)	64 bit
Bool_t	CHAR	BOOL	CHAR	
Char_t	CHAR	TEXT	CHAR	
Char_t*	CHAR(n) n<4	CHAR(n)	CHAR(n)	n < 4
Char_t*	TEXT	TEXT	VARCHAR(4000)	n > 3
string	TEXT	TEXT	VARCHAR(4000)	n > 3

Table 4.1: C++ data types to database column type mappings for MySQL, PostgreSQL and ORACLE database engines.

Some good practice to save space as far as C++ data type to SQL column type mapping is concerned:

- avoid (if possible) the long 64-bit data representation. Use it only if such a precision is required.

⁶ It is still possible to store an ordered collection of data values without conversion to a table by creating as many column as data values stored in an array. This gives the advantage to be able later on to select via extended SQL queries on a single data value from the array. The disadvantage is that large arrays produce large table and the corresponding I/O is suboptimal. In addition the user can quickly reach the internal table column-size limit of the database.

- select the fixed-length character string CHAR(n) for character string containing 3 or less characters and select always the smallest capacity for integers.
- for portability reason avoid using unsigned values for database column type.

4.3.3 Data Conversion

Once the relational table layout for the parameter container is defined (see section 4.2), the user should also define the way each data members will be streamed i.e how the data members

- get stored in their corresponding table column field type to form a table row.
- get filled from the fetched table row in the database.

As shown in Figure 4.3, the virtual database base table mapper class `FairDbObjTableMap` defines a set of virtual functions that need to be overloaded by the concrete parameter container derived class.

The first method that need to be implemented is `CreateObjTableMap()`. This method is used internally by the virtual database SQL I/O mechanism to fill results tables. The implementation is straightforward since it needs just to forward the concrete instance of the parameter class.

```
virtual FairDbObjTableMap* FairDbTutPar()::CreateObjTableMap()
{
return new FairDbTutPar();
}
```

The virtual database is doing its own parameter object pointer bookkeeping so that no memory leaks can be created during the I/O process.

The next methods corresponds to the basic SQL I/O operations on the data members

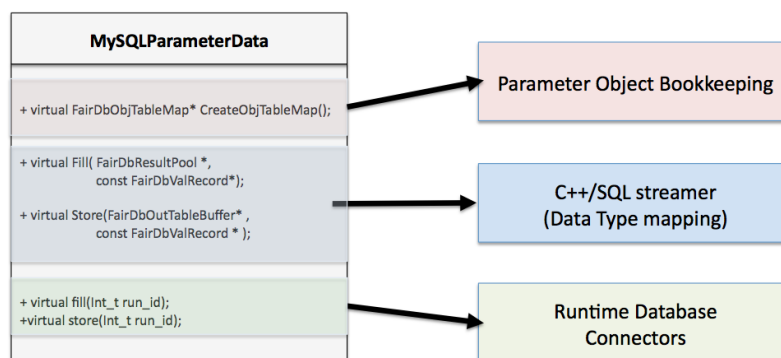


Figure 4.3: SQL-aware parameter class virtual functions responsible for parameter object pointer bookkeeping, C++/SQL data type mapping and I/O and runtime database connectors.

of a parameter container class and are implemented by the two following virtual methods respectively:

```
FairDbTutPar()::Store(FairDbOutTableBuffer*, const FairDbValRecord*);

FairDbTutPar()::Fill(FairDbResultPool*, const FairDbValRecord*);
```

These I/O methods will be described in the case of single and user-defined data type in the following sections using the example parameter class `FairDbTutPar` and `FairDbTutParBin` provided in the `Tutorial5` example subdirectory.

4.3.4 Single Data types

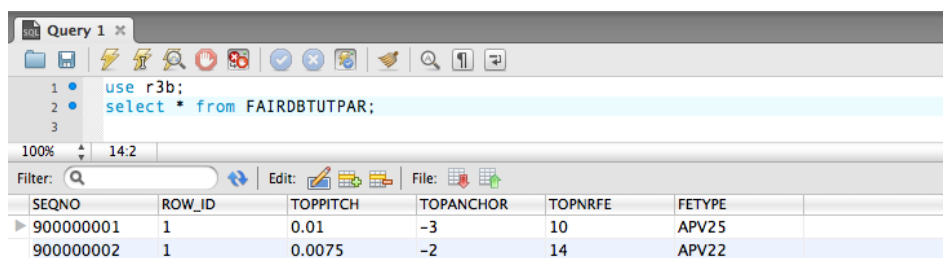
An example of C++ single data types conversion using the `Store()` virtual function is given for the `FairDbTutPar` parameter container class:

```
FairDbTutPar()::Store( FairDbOutTableBuffer* res_out,
                      const FairDbValRecord* vr)
{
res_out << fTopPitch << fTopAnchor << fTopNrFE << fFeType;
}
```

The first argument of the `Store()` function is the class `FairDbOutTableBuffer` which acts like a standard output stream. As in standard C++, the insertion operator (`<<`) which is re-programmed to properly convert C++ data type to corresponding SQL data types, is used to fill the parameter class corresponding relational table with the data members values. The user should be aware that the order in the outstreaming sequence should correspond 1:1 to the column field type sequence in the table.

One call to the `Store()` function adds one row in the relational table `FAIRDBTUTPAR` corresponding to the parameter class `FairDbTutPar`. Figure 4.4 shows two rows added to the `FAIRDBTUTPAR` table after calling two times the `FairDbTutPar::Store()` function with different set of data members values.

The class `FairDbTutPar` implements also the reverse SQL I/O operation i.e to fill the



SEQNO	ROW_ID	TOPPITCH	TOPANCHOR	TOPNRFE	FETYPE
900000001	1	0.01	-3	10	APV25
900000002	1	0.0075	-2	14	APV22

Figure 4.4: Using two calls to the `FairDbTutPar::Store()` with different set of data members values, two rows are added to the `FAIRDBTUTPAR` table.

parameter container data members with the proper values taken from the corresponding relational table stored in the database system:

```

FairDbTutPar()::Fill( FairDbResultPool* res_in,
                    const FairDbValRecord* vr)
{
res_in » fTopPitch » fTopAnchor » fTopNrFE » fFeType;
}

```

4.3.5 User-Defined Data types

User-defined data types can not be handled directly by the specific input/output SQL stream and need to be first converted into a formatted set of bytes which will be handled as compressed binary string by the virtual database. This intermediate data conversion is the responsibility of the generic stream class `FairDbStream` which distinguishes between two cases:

- In the case of static arrays, enumerations or any standard C data structure, the `FairDbStream` compresses the data using its own binary conversion mechanisms.
- In the case of user-defined class derived from the basic `TObject`, `FairDbStream` uses internally the ROOT object streaming mechanism (`TBuffer`) to serialize the object which could benefit of the additional ROOT schema evolution.

An example of User-defined single data types conversion using the `Store()` virtual function is given for the `FairDbTutParBin` parameter container class. This class defines additionally the following user defined data members:

```

class FairDbTutParBin : public FairParGenericSet
{
// Single data types

// Strip pitch on top wafer side
Double_t fTopPitch;
// Anchor point of top strip 0
Double_t fTopAnchor;
// Number of Frontend attached to top wafer side
Int_t fTopNrFE;
// Frontend type name
std::string fFeType;

```

```

// User-defined data types

// Array of Int_t fixed values
Int_t fMyIArray[3];
// Array of Double_t fixed values
Double_t fMyDArray[10];
// Root Histogramm
TH1F* fMyHisto;
...
}

```

The `Store()` virtual method uses first the service of `FairDbStream` for data compression and then apply the insertion operator `<<` on the generated binary strings to send bytes to the output stream:

```

FairDbTutParBin()::Store( FairDbOutTableBuffer* res_out,
                          const FairDbValRecord* vr)
{
// Data binary conversion
FairDbStreamer dbIArray(fMyIArray,3);
FairDbStreamer dbDArray(fMyDArray,10);
FairDbStreamer dbHisto(fMyHisto);

// Insert single data types values directly and user-defined data types
// previously converted binary streams to the output stream
res_out << fTopPitch << fTopAnchor << fTopNrFE << fFeType
<< dbIArray << dbDArray << dbHisto;
}

```

How the user defined data members are converted and store as binary strings when using the `Store()` method is shown in Figure 4.5. The table editing is done using the MySQLWorkBench SQL Editor available for MySQL database. For retrieving data stored

SEQNO	ROW_ID	TOPPITCH	TOPANCHOR	TOPNRFE	FETYPE	MYIARRAY	MYDARRAY	MYHISTO
900000001	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002300001400001f600074000...
900000002	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002300001400001f600074000...
900000003	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002300001400001f600074000...
900000004	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002300001400001f600074000...
900000005	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002300001400001f600074000...
900000006	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	400002330001400001f900074000...
900000007	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	4000023e00014000020400074000...
900000008	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	4000023e00014000020400074000...
900000009	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	4000024b00014000021100074000...
900000010	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	4000023e00014000020200074000...
900000011	1	0.01	-3	10	APV25	0000000a000...	3fb99999999...	4000023c00014000020200074000...

Figure 4.5: Contents of the table `FAIRDBTUTPARBIN` corresponding to the parameter class `FairDbTutParBin` containing user-defined data types data members. The Table editing is done from MySQLWorkBench GUI, the binary strings are represented in hexadecimal format.

as binary stream in the table `FAIRDBTUTPARBIN`, the user needs to implements a dedicated `Fill()` function that again uses the `FairDbStreamer` services.

```

FairDbTutParBin()::Fill( FairDbResultPool* res_in,
                        const FairDbValRecord* vr)
{
// clear the data contents

clear();
FairDbStreamer dbIArray(fMyIArray,3);
FairDbStreamer dbDArray(fMyDArray,10);
FairDbStreamer dbHisto(fMyHisto);

res_in » fTopPitch » fTopAnchor » fTopNrFE » fFeType
» dbIArray » dbDArray » dbHisto;

// Update data members
dbIArray.Fill(fMyIArray);
dbDArray.Fill(fMyDArray);
dbHisto.Fill(fMyHisto);
}

```

A good practice when implementing the `Fill()` function is to always clear the contents of all data members class before filling them with new fetched values from the database. In the above example of the `FairDbTutParBin::Fill()`, one can distinguish 3 main actions done in the body of the function i.e:

- At the first place via the `clean()` setting initial values to the different data members.
- A dedicated action is done for user-defined data members i.e using the `FairDbStreamer` class, all buffers corresponding to the different data members are created and their correct size in Bytes are estimated. In the example, the buffer are `dbIArray`, `dbDArray` and `dbHisto` corresponding to the static `integer` and `double` arrays and the ROOT histogram pointer `TH1F*` respectively.
- Once buffer are created in memory, the concrete streaming from table to all C++ data types is done by acting the overloaded C++ extraction operator `»` on all data members in order exactly the same way as for single data types.
- Finally the conversion of the sequence of bytes stored in the buffer to the concrete user-defined data types is achieved using the method `FairDbStreamer::Fill()`

Using the overloaded insertion and extraction operator on the dedicated input/output `FairDb` stream classes which are `FairDbOutTableBuffer` and `FairDbResultPool` respectively and combining these basic I/O operations with the `FairDbStreamer` services allow the user to stream basically any possible data-types.

One should nevertheless be aware of consequence when misusing the binary streaming feature offered by the database system.

4.3.6 Storing Large Object in a Database?

It is a good practice to avoid filling tables with huge and complicated objects streamed as binary strings or BLOB. Indeed the concept of any database is generally suited to store a

very large number of object that are relatively small in size. On the other hand, file systems are generally more suited to handle large binary data.

The decision of whether or not to store data as BLOBs must not be done based on what the DBMS offers (even if it is Oracle). The decision should instead be based on the performance requirements, scalability needs, ease of maintenance, cost of future features to be added to the application etc., and not on what seems to be more comfortable using.

One has to understand the argument of why storing BLOBs in file system is generally a wiser decision especially in a data-intensive environment.

- First, typically if dealing with huge binary data the user can query the database first for the meta-data elements (i.e paths locating files) and then for each binary data payload request made later, simply use the data location and the file system to serve the request. If, however, data is stored as BLOBs, then each binary data request will need to communicate with database to get the contents of the requested data. This can put unnecessary load on the database and increase I/O contention on the database server. As a consequence the database server is doing a lot of I/O activity and plans to keep disk I/O to minimum is disrupted. The cost of opening files containing large data is certainly less than accessing the same data from database.
- Second, for the database administrator point of view, unless the BLOB is kept in a separate table, say `geometry_blob`, from the main table `geometry`, mixing BLOB with simple column field types in the same table is going to make schema changes a nightmare. The DBA will be performing migrations on an additional terabytes and upgrading the database to a new version or to another one becomes a significantly more complicated process.
- Finally, there is the whole issue of **fragmentation of database storage** compared to file system storage. Not to mention making a mess by mixing sequential I/O and random I/O instead of separating them and benefiting from the separation. This is not just an issue with *cheap* open source database systems like MySQL. Intrinsically filesystems have better fragmentation handling than databases setting the break-even point ⁷ down from about 1MBytes to about 256 KBytes.

4.4 Advanced SQL I/O Features and Optimizations

We have seen that the overloaded virtual `Fill()` and `Store()` functions define the low-level SQL-IO operations needed for the C++ parameter object to table mapping.

Simple examples of filling implementation in both of single data types and user-defined data types have been described in the previous sections.

Nevertheless the `Fill()` method can be implemented in a more sophisticated way and, for that purpose, the user can benefit from the services that are provided by the input stream `FairDbResultPool` i.e

```
UInt_t FairDbResultPool::GetCurrentColumnName() const;
UInt_t FairDbResultPool::GetCurrentColumnNo() const;
UInt_t FairDbResultPool::GetNumOfColumns() const;
```

⁷ The break-even point defines when accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record. Basically, BLOBs smaller than 256KB are more efficiently handled by a database, while a filesystem is more efficient for those greater than 1MB. Of course, this will vary between different databases and filesystems.

These functions gives additional meta-data information about the table being accessed to the user i.e the total number of columns in the row , the name of the current column being accessed and its number ⁸.

The user can get the information about the type and size of the data defined in a particular column of the table being accessed:

```
FairDbFieldType FairDbResultPool::GetCurrentColumnFieldType() const;
```

The user can create an instance its own column field using the defined FairDb standard data types enumeration structure (FairDb::DataTypes):

```
FairDbFieldType fUserFieldType(FairDb::kChar);
```

In the example above the user defines a C++ data type `char` and can for instance compare it with the type obtained from the current column in the table row being accessed.

For this purpose, the class `FairDbFieldType` provides the following test functionality features on the column field types i.e

- Capacity (data size) test:

```
Bool_t FairDbFieldType::IsSmaller(FairDbFieldType& aType);
```

- Type equivalence (same field type) test:

```
Bool_t FairDbFieldType::IsEquivalent(FairDbFieldType& aType);
```

The user can benefit from the class `FairDbFieldType` services to implement a more selective `Fill()` functions. For instance, the user can modify the original `FairDbTutPar::Fill()` function if a precise control on the names, numbers and types of the columns of the table being accessed is required:

```
FairDbTutPar()::Fill( FairDbResultPool* res_in,
                    const FairDbValRecord* vr)
{
// Get number of Columns
Int_t n_cols = res_in.GetNumOfColumns();

// Since The first column has been processed
// iterate on the next ones from index=2
for (Int_t cur_col = 2; cur_col <= n_cols; ++cur_col)
{
string column_name = res_in.GetCurrentColumnName();
if ( column_name == "TOPPITCH" ) res_in » fTopPitch;
else if ( column_name == "TOPANCHOR" ) res_in » fTopAnchor;
else if ( column_name == "TOPNRFE" ) res_in » fTopNrFe;
else if ( column_name== "FETYPE" ) res_in » fFeType;
else {
std::cout << "-I- Unknown column found at index: " << cur_col
<< "column name: " << col_name << std::endl;
res_in.GetNextCurrentColumn();
}
}
```

⁸NB: the column numbering starts at one

Note that if the table being accessed contains many rows valid at time, such a data stream control using a string comparison on column names could be performance inefficient.

In the case the table column layout is fixed, it is always better in terms of performance to take the data as it comes and use the insertion on an ordered sequences of data members as implemented in the original `FairDbTutPar::Fill()` function ⁹.

The user can benefit of further services that `FairDbResultPool` offers:

```
string FairDbResultPool::GetTableName();
Bool_t FairDbResultPool::IsComplete() const;
UInt_t FairDbResultPool::GetCurrentRowNo() const;
```

which returns the name of the table being accessed, whether there is or not data left to extract and the current row number.

4.4.1 Table Schema Evolution and Context Selection

The next example presents a more complicated implementation of the same `Fill()` function showing how to test on different table layouts (table schema evolution) and to compare fetched values from the database table and the context information used by the query.

```
FairDbTutPar():Fill( FairDbResultPool* res_in,
                   const FairDbValRecord* vr)
{
    string det_name;

    Int_t run_id;

    // Count the number of data columns
    //(skipping SEQ_ID and ROW_ID first column)

    Int_t n_cols =
        res_in.GetNumOfColumn()-(res_in.HasRowId()?1:0)-1;
```

⁹ The user is not forced to define a strict 1:1 correspondence between database column in a table and the class data members. As long as the `Fill` and `Store()` implementations are consistent and satisfy its client it can store either the full information or just a filtered information from the database table in order to fill the persistent data members.

```
// 1) Tests on different table layout

// Table layout with only 1 column
if ( 1 == n_cols ) {
res_in » det_name;
}
// Table layout with 2 columns
else if ( 2 == n_cols ) {
res_in » det_name » run_id;
}
else {
std::cerr « "Table " « GetName() « " number columns " « n_cols
« "has different layout than 1 or 2 column"
« std::endl;
}
}
```

```
// 2) Comparison between Query Context Info
// and fetched data (det_name) from Table

Int_t det_id = ((vr) ? vr->GetValInterval().GetDetectorId() : 0);
Int_t data_id = ((vr) ? vr->GetValInterval().GetDataId() : 0);

//Get detector ID from the detector name
Int_t fetched_det_id = Detector::CharToEnum(det_name.c_str());
if ( fetched_det_id != det_id)
cerr « "Detector column ' " « det_name « "' (" « fetched_det_id
« ") did not match DetectorId (" « det_id « ")" « endl;
} //! End of the Fill() method
```

In the first part, the user tests on different layouts of the database table which could have 1 or 2 columns storing the following data:

- The detector name as (**string**) in the first column.
- The run number as (**int**) in the second column

Eventhough the table exists with two different layouts in the database, the user is testing on the number of columns obtained using the `FairDbResulPool::GetNumOfColumn()` method and according to the number of columns found extracts either only the detector name or both detector name and run number.

In the second part the fetched detector name `det_name` is converted into its corresponding detector identifier `fetched_det_id` and directly compared with the detector identifier from the context used by the query. If detector identifiers are not equal the data fetched from the database table is inconsistent with the context used by the query.

4.4.2 Caching Activation

As explained in section 3.4, a global caching mechanism is activated from the runtime database SQL-IO interface using its method `FairParTSQLIo::SetCache(TString& dir)`. Once activated the caching mechanism allows fetching data from the database table faster by

caching the query results on the file system as disk files. Eventhough caching is activated globally using the `FairParTSQLIo` interface, the user can also activate or deactivate the caching mechanism for each parameter container object overriding the `CanCache()` method. For instance, to activate the caching mechanism related to `FairDbTutPar`:

```
Bool_t FairDbTutPar::CanCache() const { return kTRUE;}
```

and the same way to deactivate it.

```
Bool_t FairDbTutPar::CanCache() const { return kFALSE;}
```

Note that overriding the virtual method `FairDbObjTableMap::CanCache()` has priority on the global setting done using `FairParTSQLIo::SetCache(TString& dir)`.

4.4.3 Ordering Rows

As explained in section 3.6, it is possible to force the virtual database to order the results set according to the `row_id` column index. Since the `row_id` column index is incremented everytime a row is written with the same context information to the same table, the ordering of rows in a table is determined by the way the data is written to the database. If the data writing process sequence does not form a natural way to access it, parameter object can define their own indexing scheme overriding the virtual method `FairDbObjTableMap::GetIndex(UInt_t)`.

For instance

```
UInt_t FairDbTutPar::GetIndex(UInt_t) const { return fTopAnchor/2.; }
```

just meant to show how a parameter object can implement its own indexing scheme from some identification number.

To build such a index some rules should be followed:

- Index word should be a 4 bytes size word.
- Must be unique in the set
- Index should be a unsigned integer as the sign bit has no relevance. Nevertheles signed integer index are allowed.

Versioning Management

This chapter describes the handling of condition data within the virtual database context i.e the retrieval and storage of parameter data using a validation scheme.

The principle of the versioning management or validation scheme has been introduced in section 1.8. It is based on the fact that every database transactions for reading or writing from/to a payload data table is done first by accessing the meta-data from the corresponding auxiliary validation table.

The validation table contains the the full information context used by the user to store the parameter data. Nevertheless it is not necessary for the user when accessing the data to specify the full set of validation information.

In fact, the minimal specification of what data to return requires no more than the table name or class name, a dedicated timestamp, which detector and whether it was simulation or real data.

5.1 Validation Table

Each database table created within the virtual database context by overloading the function

```
FairDbObjToTableMap::GetTableDefinition(const char* table_name)
```

is automatically associated with its correponding validation table as explained in section 4.2.2.

In the case of the example parameter container `FairDbTutPar`, executing the function `FairDbTutPar::GetTableDefinition()` to create the database table will trigger automatically the creation of the corresponding validation table if this does not already exist.

It means that internally the virtual database commits the following SQL statement:

```
create table FAIRDBTUTPARVAL(
    SEQNO integer not null primary key,
    TIMESTART datetime not null,
    TIMEEND datetime not null,
    DETID tinyint(4),
    DATAID tinyint(4),
    VERSION integer,
    COMPOSITEID integer,
    TIMEINCR datetime not null,
    TIMETRANS datetime not null,
    key TIMESTART (TIMESTART),
    key TIMEEND (TIMEEND));
```

The creation of the validation table is only triggered if it does not exist previously i.e it has not been previously created by the user. Indeed, it is always possible for the user to create

the validation table using the following code:

```
FairDbConnectionPool*
fConnPool = FairDbTableInterfaceStore::Instance().GetConnectionPool();

// Select first database entry
Int_t dbEntry = 0;

// Create a unique statement on choosen DB entry
auto_ptr<FairDbStatement> stmtDb(fConnPool->CreateStatement(dbEntry));

if ( ! stmtDb.get() ) {
cout << "-E- FairDbTutParBin::Store() Cannot create a statement for
Database_id: " << dbEntry << endl;
exit(1);
}

stntDbn.Commit( " create table if not exists FAIRDBTUTPARVAL ("
                " SEQNO int not null primary key,"
                " TIMESTART datetime not null,"
                " TIMEEND datetime not null,"
                " DETID tinyint,"
                " DATAID tinyint,"
                " VERSION int,"
                " COMPOSITEID int,"
                " TIMEINCR datetime not null,"
                " TIMETRANS datetime not null,"
                " key TIMESTART (TIMESTART),"
                " key TIMEEND (TIMEEND) )");
```

Eventhough it is not required for the main cases, the payload data table can also be directly created by the user using the `FairDbStatement::Comit()` function with the correct associated SQL statement as argument:

```
stmtDb.Commit( "CREATE TABLE IF NOT EXISTS FAIRDBTUTPAR ("
               " SEQNO INT NOT NULL,"
               " ROW_ID INT NOT NULL,"
               " TOPPITCH DOUBLE,"
               " TOPANCHOR DOUBLE,"
               " TOPNRFE INT,"
               " FETYPE TEXT, "
               " PRIMARY KEY(SEQNO,ROW_ID) )");
```

Figure 5.1 shows a snapshot of the `FaiDbTutPar` validation table (`FAIRDBTUTPARVAL`) once created in the database system. Each row in the validation table corresponds to a successful insertion of values in the corresponding payload table. During data retrieval, a condition defining an interval of validity (timestamp), a detector and a data type is required by the interface.

Then, the virtual database fetches the valid data that corresponds to this specific condition:

- Using a fast search algorithm, it matches the given condition to a row (entry) in the correct Validation Table and get the corresponding sequence number.
- The sequence number as unique row identifier in the Validation Table is used as a key into the main payload data table to get all the relevant rows matching that key.

SEQNO	TIMESTART	TIMEEND	DETID	DATAID	VERSION	COMPOSITEID	TIMEINCR	TIMETRANS
900000001	2013-10-22 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-22 14:27:54
900000002	2013-10-22 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-22 14:27:54
900000003	2013-10-22 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-22 14:27:57
900000004	2013-10-22 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-22 14:27:57
900000005	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:03:14
900000006	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:03:14
900000007	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:07:28
900000008	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:07:28
900000009	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:08:47
900000010	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:08:47
900000011	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:10:37
900000012	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:10:37
900000013	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:37:50
900000014	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:37:50
900000015	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:53:01
900000016	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:53:01
900000017	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 07:55:34
900000018	2013-10-23 ...	2038-01-19 ...	8	1	1	-1	2013-10-22 ...	2013-10-23 07:55:34
900000019	2013-10-23 ...	2038-01-19 ...	8	1	0	-1	2013-10-22 ...	2013-10-23 08:58:00

Figure 5.1: Snapshot of the FairDbTutPar validation table using MySQLWorkbench. Each validation interval corresponds to a row of the validation table FAIRDBTUTPARVAL. The row is given a unique sequence number SeqNo which in turn acts as a key to access the corresponding payload data rows in the main data table FAIRDBTUTPAR

In the next sections, writing and reading parameter data container using the versioning management mechanism will be described.

5.2 FairParTSQLIo Responsibilities

In section 3.2 the main SQL I/O manager class FairParTSQLIo services corresponding to the general database client settings such as connection handling, caching mechanism, have been described.

Within the context of the FairRoot runtime database, the SQL I/O manager class FairParTSQLIo plays a central role in the framework initialization scheme. The FairRoot framework initialization scheme supposes that every new data file opened is uniquely identified by its **run number**. Within the runtime database context, only the run number is used to store or retrieve data.

5.2.1 Run Numbers versus Timestamps

In section 1.8 it has been described that the key aspect of the virtual database is that decisions are based on time and not such things like run number. Nevertheless the FairParTSQLIo can accommodate the use of run number converting it internally to a timestamp. The run number conversion depends on the data type

Simulation Data: the run number is automatically generated using the run number generator class FairRunIdGenerator. The run number generator use internally the unix

EPOCH time definition struct `timespec` ¹⁰

```
struct timespec
{
time_t tv_sec; // seconds
long tv_nsec; // nanoseconds
}
// time_t seconds is relative to Jan 1, 1970 00:00:00 UTC
```

to map the beginning time of a simulation run into a 32 bit `unsigned int`. In the writing process the run number generator class `FairRunIdGenerator` is used the following way:

```
// Generate a unique RunID
FairRunIdGenerator runID;
UInt_t runId = runID.generateId();
```

In the reading process the run number can be created from a given timestamp using the `FairDB ValTimeStamp` services:

```
// Define the data in the format (YYYY,MM,DD,hh,mm,ss)
ValTimeStamp tStamp(2013,10,22,15,00,00);
// Map the date into a 32 bit unsigned integer
UInt_t runId = tStamp.GetSec();
```

Real Data: In this case one can not assume that the conversion to time is possible. If the run numbering of an experiment do not easily relate to time definition, a run number to timestamp relation has to be created as a map and stored as a table in the database. For instance a run number can be associated to two times defining the run period: `run_start_time` and `run_stop_time`.

5.2.2 Runtime Database and SQL I/O Interface

This section describes the interplay between the virtual database and the runtime database within the `FairRoot` framework.

It has been described above that within the runtime database framework, writing and retrieving parameter data always suppose that a run number is given.

Figure 5.2 shows a simplified sequence diagram of a parameter writing process in `FairRoot`. Each parameter container created during a typical `FairRoot` application is stored internally in the main parameter manager class `FairRuntimeDb` in a list. For writing the data contents of a parameter container to any I/O system (text-based files , ROOT files or database system) the corresponding I/O interface has to be instantiated and set as a possible output from the parameter manager class `FairRuntimeDb`.

For instance if the user wants to use a database system as output

¹⁰ The use of `time_t` (and its default Unix implementation as a 32 int) implies overflow conditions occur somewhere around Jan 18, 19:14:07, 2038. If this implementation is not changed in the Unix kernel when it becomes significant a manual intervention will be necessary.

```
// Create the Runtime Database ( parameter manager class )
FairRuntimeDb* db = FairRuntimeDb::instance();

// Create an instance SQL based I/O
FairParTSQLIo* out_sql_io = new FairParTSQLIo();

// Set SQL based I/O as output db->setOutput( out_sql_io);
```

Once the SQL I/O interface `FairParTSQLIo` is set as output, the user can at any time trigger a writing of each parameter stored internally in the main parameter manager class `FairRuntimeDb` using :

```
FairRuntimeDB::writeContainers();
```

As shown in Figure 5.2, the `FairRuntimeDB::writeContainers()` method calls in turn the `FairDetParTSQLIo::write(FairParSet*)` function which forwards the run number information to the concrete implementation of the overloaded `FairParSet::store(Int_t runId)` virtual method.

The virtual database will then store the data to the user defined database system and timestamp it using the run number information forwarded to the detector SQL I/O interface `FairDetParTSQLIo`.

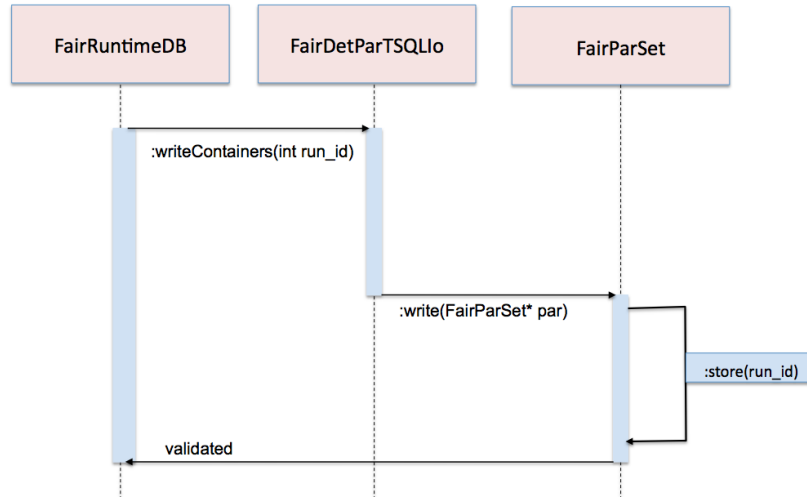


Figure 5.2: Simplified sequence diagram showing the classes and method involved during parameter writing using the virtual database. The concrete SQL I/O is done by overloading the virtual function `FairParSet::store(int runId)` which use the forwarded run number from the runtime database as argument.

On the other hand, if the user wants to use a database system as input source for parameter initialization:


```
// Create the Runtime Database ( parameter manager class )
FairRuntimeDb* db = FairRuntimeDb::instance();

// Create an instance of SQL based I/O
FairParTSQLIo* in_sql_io = new FairParTSQLIo();

// Set SQL based I/O as first input db->setFirstInput( out_sql_io);
```

Once the SQL I/O interface `FairParTSQLIo` is set as first input (or eventually second), the user can at any time trigger a initialization of each parameter stored internally in the main parameter manager class `FairRuntimeDb` using :

```
FairRuntimeDB::initContainers(Int_t run_id);
```

As shown in Figure 5.3, the `FairRuntimeDB::initContainers(Int_t run_id)` method calls in turn the `FairDetParTSQLIo::init(FairParSet*)` function which forwards the run number information to the concrete implementation of the overloaded `FairParSet::fill(Int_t runId)` virtual method.

The virtual database will then store the data to the user defined database system and timestamp it using the run number information forwarded to the detector SQL I/O interface `FairDetParTSQLIo`. Once the SQL based I/O interface is set as input/output of

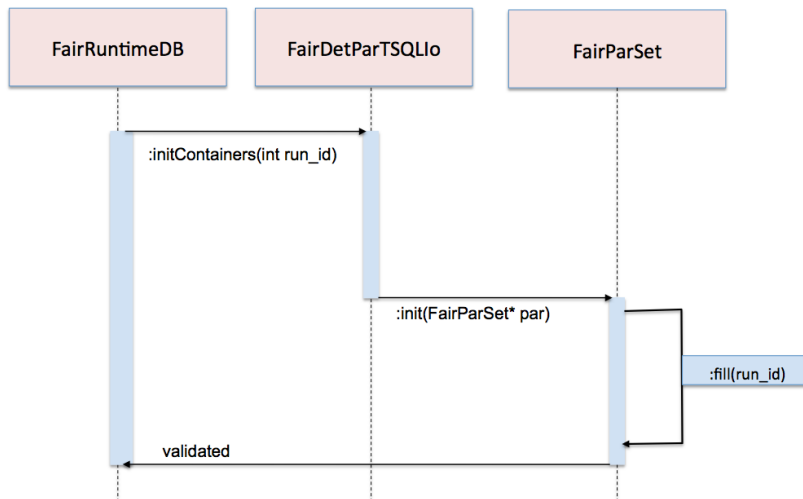


Figure 5.3: Simplified sequence diagram showing the classes and method involved during parameter initialization using the virtual database. The concrete SQL I/O is done by overloading the virtual function `FairParSet::fill(int runId)` which use the forwarded run number from the runtime database as argument.

the runtime database, all mechanism described above are done automatically. Only the parameter data I/O methods for writing and reading respectively:

```
FairParSet::store(Int_t run_id)
FairParSet::fill(Int_t run_id)
```

have to be implemented for each parameter container. The next sections will describe in details how to prototype such I/O functions in the parameter class.

5.3 Templated I/O

Within the virtual database context, both parameter data intrinsic SQL I/O and versioning management mechanisms are combined and implemented in a generic way within dedicated reader and writer C++ templates.

In the virtual database design, the usage of template metaprogramming technique to implement a dedicated SQL I/O mechanism has been chosen considering the following advantages:

Genericity: templated I/O allows for automatic generation of stream SQL I/O primitives even for complex user-defined C++ parameter objects. Using templates, the virtual database I/O system acts as a parser-based I/O primitives generator and can be considered as a generic object SQL I/O library.

Non-intrusive I/O mechanism: The reader or writer templates define SQL I/O primitives using any user-defined parameter container class as a generic data type. The user keeps the original class declaration for application development without requiring the classes to be modified for SQL I/O. The only restriction for the developer is to derive the parameter class from the framework base parameter class `FairParSet` and overload the relevant virtual functions which are used by the reader and writer templates implementation.

Maintenance: The fact that all stream SQL I/O primitives are implemented in a generic way by only one reader and one writer template eases the task of managing application I/O problems i.e. the I/O related source code is well localized and I/O problems are easier to debug.

5.3.1 FairDbWriter Template

In order to write the persistent data members of a parameter container object, the user needs the services of the writer template `FairDbWriter`.

Let's take as an example the parameter class `FairDbTutPar` from the `tutorial5` example. To write the `FairDbTutPar` data members to any database the user needs to do an explicit instantiation of the templated class `FairDbWriter` using the parameter class `FairDbTutPar` as type argument:

```
FairDbWriter<FairDbTutPar> aWriter;
```

Note that the writer template `FairDbWriter` only writes to already created tables in the database. It is the responsibility of the user to ensure before any writing to the database that the table supposed to be filled really exists. In the next section an example will show how to check for the existence of a particular table within the database system and, if the table does not exist, how to create the table within the same job as the one that fills.

Once the template is explicitly instantiated, a specific filling of the relational table including the underlying version management can be activated by first creating a *Interval Of Validity*

using the `ValInterval` class services. The user should decide the validity interval of the data to be written and store it in a `ValInterval` object:

```
// Define start and stop time for interval
ValTimeStamp t_start(2013,10,22,15,00,00);
// Add to t_start 15 days to get a t_stop
ValTimeStamp t_stop = t_start.GetSec() + 15*24*60*60;

// Create an Interval of Validity (IOV)
ValInterval iov( Detector::kGfi,
                Datatype::kData,
                t_start,
                t_stop,
                "example");
```

The `ValInterval` object defines precisely the *Interval Of Validity* for the subsequent filling procedure i.e

- The data validity time interval is `[t_start, t_stop]`
- The data is related to fiber tracker (GFI) (`Detector::kGfi`). The detector numbering (enum) is defined by the user in the header file `$(VMCWORDIR)/input/db/db_detector_def.h`.
- The data type is real data (`Datatype::kData`)

Once the *Interval Of Validity* is defined, it is given to the writer template `FairDbWriter` as first argument in order to activate the I/O mechanism within the *Interval Of Validity* context.

```
FairDbWriter<FairDbTutPar> myWriter;
myWriter.Activate(iov);
```

By default the filling of the `FAIRDBTUTPAR` will be done in the first database defined in the database pool. It is possible to change the defaults adding more arguments to the `Activate()` method. These additional arguments will be described later.

```
// Create parameter object par0
FairDbTutPar par0;
par0.SetTopPitch(1.2);
par0.SetTopAnchor(15.6);
par0.SetNrTopFE(12);
par0.SetFeType("par0");
```

```
// Create parameter object par1
FairDbTutPar par1;
par1.SetTopPitch(1.2);
par1.SetTopAnchor(15.6);
par1.SetNrTopFE(12);
par1.SetFeType("par1");

// Now store the example data as rows
myWriter << par0;
myWriter << par1;
```

Each insertion operation will create a separate row in the FAIRDBTUTPAR table with the corresponding data previously set in `par0` and `par1` parameter objects.

The generic insertion operator "`<<`" i.e

```
FairDbWriter<T>& operator<<(const T& row);
```

will internally execute the `FairDbTutPar::Store()` method, explained in details in section 4.3.4 in order to do the class related data member SQL intrinsic I/O:

```
FairDbTutPar()::Store( FairDbOutTableBuffer* res_out,
                      const FairDbValRecord* vr)
{
res_out << fTopPitch << fTopAnchor << fTopNrFE << fFeType;
}
```

Furthermore, during insertion operation the writer template checks if the type of the data to be written is compatible with the database table defined field. If this basic check fails an error message is sent and the data is registered as invalid and will not be outputted anymore. Additionally the IOV used for the filling will be stored as meta-data in the auxiliary validation table FAIRDBTUTPARVAL.

The data payload added rows corresponding to `par0` and `par1` objects of the FAIRDBTUTPAR table will be linked to the corresponding rows in the table FAIRDBTUTPARVAL via automatically generated primary key (`seqNo`).

Finally to end properly the filling procedure, the user should use the `Close()` from the writer template:

```
aWriter.Close();
```

Note that the corresponding transaction for filling the table can only be committed if the user calls the `Close()`. Before that call, only the sequence of corresponding SQL statements are internally stored by the virtual database in the SQL processor queue in pending status ready to be executed. All is done as if the writer template class `FairDbWriter` always caches the data but performs no database I/O until the `Close()` command is issued.

At any time before issuing the `Close()` command, the SQL I/O can be internally stopped and the internal SQL statements cache cleared either by destroying the template class itself or by using the method:

```
aWriter.Abort();
```

Such an implementation of a SQL I/O queuing the SQL commands only occurs until

- All data has been collected
- The corresponding queued SQL commands are sent and executed on the database server all together at a time

Using the two steps implementation has the advantage of minimizing the probability of writing wrong data. For instance if the user suddenly finds out a problem with some calibration data while the write template `FairDbWriter` is assembling it, he or she can always use the `FairDbWriter::Abort()` method to completely cancel the I/O.

The same way if the writer template `FairDbWriter` detects a mistake while processing the user data, it will not perform any data output when the method `FairDbWriter::Close()` is issued.

Note that eventhough the writer template is a lightweight object when just instantiated, it has the tendency of increasing its memory size as internally the table rows are cached. For this reason, it is crucial to always explicitly **delete** the `FairDbWriter` writer template once the I/O is completed otherwise it can lead to an important memory leak ¹¹.

5.3.2 FairDbReader Template

The same way as for the writing procedure, in order to access the data from the existing tables stored in the database the user needs the services of the reader template `FairDbReader`. Lets again take as an example the parameter class `FairDbTutPar` from the `tutorial5` example.

To access the `FairDbTutPar` data from any database the user needs to do an explicite instantiation of the templated class `FairDbReader` using the parameter class `FairDbTutPar` as type argument:

```
FairDbReader<FairDbTutPar> aReader;
```

To activate the read I/O operation, the user need to provide a *condition* for the query which defines

- The date time corresponding to the current run or better, event number
- Detector and data type.

```
// Get run number from FairRuntimeDb
Int_t rid =
FairRuntimeDb::instance()->getCurrentRun()->getRunId();

// Create a TimeStamp associated to rid
// (conversion from run number to time )
ValTimeStamp ts(rid);

// Define a Condition
ValCondition condition(Detector::kGfi,DataType::kData,ts);
```

¹¹ To minimize the probability of a memory leak when using the writer template `FairDbWriter` it is forbidden to assign to `FairDbWriter` object or event copy construct `FairDbWriter` object.

Once the *Condition* is defined, it is given to the reader template `FairDbReader` as first argument in order to activate the I/O mechanism for this *Condition*. Additionally if multiple version of the same parameter container exists, the user can specify the version number as second argument (version number default is 0).

```
// User Defined Parameter version number
Int_t version = 12;
// Activate reading for this Context
aReader->Activate(condition, version);
```

By default the data extraction from the table `FAIRDBTUTPAR` will be done using the first database defined in the database pool. It is possible to change the defaults adding more arguments to the `Activate()` method. These additional arguments will be described later. During extraction, the reader template `FairDbReader` execute the `FairDbTutPar::Fill()` method, explained in details in section 4.3.4 in order to do the class related data member SQL intrinsic I/O:

```
void FairDbTutPar::Fill(FairDbResultPool& res_in, const FairDbValRecord*
valrec)
{
res_in » fTopPitch » fTopAnchor » fTopNrFE » fFeType;
}
```

Nevertheless the reader template always organizes the data as rows, so that if multiple rows correspond to the condition provided by the user, the reader template fetches all valid rows caches them.

Eventually, the user needs to loop over all rows to obtain the data in its integrity i.e

```
// Get number of rows
Int_t n_rows = aReader.GetNumRows();

// Loop over all rows
for (int i = 0; i < n_rows; ++i) {
const FairDbTutPar* cgd = (FairDbTutPar*) aReader.GetRow(i);
if (!cgd) continue;
// process the row
...
}
```

From the code fragment above, one sees that accessing a row and mapping back the row data content into the parameter container object is done by the generic method:

```
const T* FairDbReader::GetRow(UInt_t row_index) const;
```

where, in the example, the generic type `T` corresponds to the parameter class type `T = FairDbTutPar`. The method returns a `const` pointer to a memory-mapped parameter object of type `FairDbTutPar` corresponding to the row identified by `row_index`. The row index `row_index` should be in the range

```
0 <= row_index <= FairDbReader::GetNumRows()
```

Note that the method `FairDbReader::GetRow()` returns a non-null pointer only if the row index is within the allowed range otherwise it returns a null pointer. Nevertheless, it is recommended to always check for valid pointer.

If the user is only interested in the last update made to the database table corresponding to the parameter container data, he/she can use the following code fragment which selects only the first row:

```
// Get number of rows
Int_t n_rows = aReader.GetNumRows();

// Select the last one
if ( n_rows > 1 ) n_rows = 1;

// Loop over the rows
// and assign fetched values to data members
for (int i = 0; i < n_rows; ++i) {
FairDbTutPar* cgd = (FairDbTutPar*) aReader.GetRow(i);
if (!cgd) continue;
fTopPitch = cgd->GetTopPitch();
fTopAnchor = cgd->GetTopAnchor();
fTopNrFE = cgd->GetNrTopFE();
fFeType = cgd->GetFeType();
....
}
```

Note that from the example above the user simply gets the data using the `FairDbTutPar` standard accessors and can re-assign them to the same data-members in order to update their values to the new ones fetched from the database.

5.4 Condition data

This section comes back to the implementation of the SQL-aware parameter `FairDbTutPar` from the `tutorial5` example directory. It shows the user how to programmatically combine all different aspects of the virtual database features previously described in order to create his/her own SQL-aware parameter class.

The `FairDbTutPar` header and implementation files are located at:

```
$(VMCWORKDIR)/example/tutorial5/src/FairDbTutPar.h
$(VMCWORKDIR)/example/tutorial5/src/FairDbTutPar.cxx
```

5.4.1 Data Members Structure

From the SQL-aware parameter class structure one can distinguish between two types of data members:

- Persistent data members corresponding to the actual parameter data.
- Transient data members corresponding to the virtual database infrastructure classes.

```

class FairDbTutPar : public FairParGenericSet
{
    ....
private:
// ***** Persistent Data Members ***** //
// Strip Data Parameters
Double_t fTopPitch; // Strip pitch on top wafer side
Double_t fTopAnchor; // Anchor point of top strip 0
Int_t fTopNrFE; // Number of FE attached to top wafer side
std::string fFeType; // Frontend type name

// ***** Transient Data Members ***** //
// Database Pool Index
Int_t fDbEntry; //!
// Parameter Container SQL Writer Meta-Class
FairDbWriter<FairDbTutPar>* fParam_Writer; //!
// Parameter Container SQL Writer Meta-Class
FairDbReader<FairDbTutPar>* fParam_Reader; //!
// Connection Pool
FairDbConnectionPool* fMultConn; //!
}

```

The first set of data members corresponds to the parameter data. The second set corresponds to the virtual database infrastructure class to handle:

Database entry : Identifier of the database system in use for SQL I/O. It corresponds to the database index in the priority list defined using the environment variable FAIRDB_TSQL_URL (see section).

Reader Template: FairDbReader<FairDbTutPar>* defined using FairDbTutPar type (see section 5.3).

Writer Template: FairDbWriter<FairDbTutPar>* defined using FairDbTutPar type (see section 5.3).

Database Connection Manager: pointer to FairDbConnectionPool manager class in order to handle client-server connections (see section 3.3).

All data members need to be initialized as usual in the class constructor. However initializing templates within ROOT needs a special care.

5.4.2 Template Instantiation

Since version 3.04 ROOT provides template support for classes up to three templates arguments.

It is also possible to *CINTify* your template to make it visible within the macro interpreter CINT ¹¹. The limitation in being CINT compatible is that each template class has to be **explicitly instantiated**, which means that a template should be initialized globally (like

¹¹ ROOT provides special `ClassDefT` and `ClassImpT` macros for classes with two or three template arguments. These special macros can be used directly even for a class template. `ClassImpT` is used to register an implementation file in a class. For class templates, the `ClassImpT` can only be used for a specific class template instance.

a static variable) with the proper type before using them in the class scope. For instance, the implementation file `FairDbTutPar.h` uses the following declarations within the `include` section:

```
#include "FairDbReader.tpl"
template class FairDbReader<FairDbTutPar>;

#include "FairDbWriter.tpl"
template class FairDbWriter<FairDbTutPar>;
```

In order to do ROOT template instantiation, the user should follow the following simple rules:

LinkDef.h For each parameter class the user must add an entry to the `LinkDef.h` file so that `rootcint` can supply the specific methods (like special `Streamer()` functions) to properly integrate the class into the ROOT framework. For example, if the user has implemented three variants of the parameter class `FairDbTutPar` the following lines should be added:

```
#pragma link C++ class FairDbReader<FairDbTutPar1>;
#pragma link C++ class FairDbReader<FairDbTutPar2>;
#pragma link C++ class FairDbReader<FairDbTutPar3>;
```

And of course the same should be done for the corresponding writer templates. Note there is an order to follow when adding `pragma` commands. Suppose you have implemented a class `MyParA` that defines a `FairDbReader<MyParB>` as data member. If `MyParA` and `MyParB` class are compiled in the same shared library then the `pragma` commands should be added in order i.e

```
#pragma link C++ class FairDbReader<MyParB>;
// preceding:
#pragma link C++ class MyParB;
```

CMakeLists.txt The additional lines to the `LinkDef.h` file tells `rootcint` what new SQL-Aware parameter classes have been introduced. Nevertheless to perform the proper compilation and library build, the user should ensure that the `CMake` system supplies `rootcint` with the proper virtual database template headers as follows:

```
set(HEADERS
FairDbTutPar.h
FairDbTutContFact.h
${CMAKE_SOURCE_DIR}/dbase/dbInterface/FairDbReader.h
${CMAKE_SOURCE_DIR}/dbase/dbInterface/FairDbWriter.h
)

set(LINKDEF FairDbTutLinkDef.h)
```

5.4.3 Data Encapsulation

This section comes back to some basic of old good OO design i.e data encapsulation which aims to hide implementation and only expose abstraction.

A SQL-aware parameter data class can be seen as a portal between a database relational table and the end users who will use the data it contains. When prototyping a parameter data class, the user can implement an exact mapping between the standard *getters* in the parameter class and the table columns in the database.

The `FairDbTutPar` parameter class is an example of such an implementation i.e to each single data member corresponds a table column and a getters function.

Nevertheless for calibration parameter data this 1:1 mapping between columns in the table and *getters* can be inappropriate because usually it takes time to find out the proper calibration procedure and which calibration constants are necessary to perform the calibration itself.

Suppose the user has to use for the calibration a 4 parameter functional. The user keeps, in a generic way, all functional parameters in the database table and the calibration method identifier

```
CalType INT,
par_0 FLOAT,
par_1 FLOAT,
par_2 FLOAT,
par_3 FLOAT,
```

Since the meaning of the parameter `par` only depends on the type of calibration identified by `CalType`, instead of *getters* for each parameter, it is far better to provide the parameter class a calibration method:

```
Double_t MyParamClass::DoCalibration(Double_t raw) const;
```

5.4.4 Storing Condition Data

This section shows how to combine the writer template `FairDbWriter` with the runtime database I/O interface in order to write condition data¹² using the runtime database API. In section 5.2.2, we have seen that the `FairRuntimeDB::writeContainers()` method calls in turn the `FairDetParTSQLIo::write(FairParSet*)` function which forwards the run number information to the concrete implementation of the overloaded `FairParSet::store(Int_t runId)`.

Consequently, the user has to implement the `FairParSet::store(Int_t runId)` using the writer template `FairDbWriter`. The following code fragment shows how the first implementation part of the `FairDbTutPar::Store()` method. It uses the services of the database connection manager class `FairDbConnectionPool` to do a basic connection test i.e if the database system located at the index defined by the parameter object `FairDbTutPar::GetIndex()` is accepting a prepared statement.

¹² Within this context, condition data means parameter data associated with a condition.

```

void FairDbTutPar::store(UInt_t rid)
{
// Boolean IO test variable
Bool_t fail= kFALSE;

// Create a unique statement on choosen DB entry
auto_ptr<FairDbStatement> stmtDbn(fMultConn->CreateStatement(GetDbEntry()));

if ( ! stmtDbn.get() ) {
cout << "-E- FairDbTutPar::Store() Cannot create statement for Database_id:
" << GetDbEntry()
<< " Please check the FAIRDB_TSQL_* environment. Exiting ... " << endl;
// Not recoverable error
exit(1);
}
}

```

If the basic check is successful, the user should test if a relational table corresponding to the parameter container class already exists within the connected database system. The check is done giving as input the user defined table name (FAIRDBTUTPAR) and using the `FairDbConnection::TableExists(string &)` services. If not, the user packs in a list the relevant SQL commands to actually create the corresponding table:

```

// Check if for this DB entry the table already exists.
// If not call the corresponding Table Definition Function

std::vector<std::string> sql_cmds;
TString atr(GetName());
atr.ToUpper();

if ( ! fMultConn->GetConnection(GetDbEntry())->TableExists("FAIRDBTUTPAR")
) {
// Pack SQLstatements into the STL vector
sql_cmds.push_back(FairDb::GetValDefinition("FAIRDBTUTPAR").Data());
sql_cmds.push_back(FairDbTutPar::GetTableDefinition());
}
}

```

Once all sql commands are queued in the list, the user executes them using the `FairDbStatement::Commit(string&)` service. In order to detect if anything went wrong during the transaction with the database, the `Bool_t FairDbStatement::PrintExceptions()` is used.

```

// Packed SQL commands executed internally via SQL processor
std::vector<std::string>::iterator
                                itr(sql_cmds.begin()),
                                itr_end(sql_cmds.end());

while( itr != itr_end ) {
std::string& sql_cmd(*itr++);
stmtDbn->Commit(sql_cmd.c_str());
if ( stmtDbn->PrintExceptions() ) {
fail = true;
cout << "-E- FairDbTutPar::Store() Error Executing SQL commands" << endl;
}
}

// Refresh list of tables in connected database
// for the choosen DB entry
fMultConn->GetConnection(GetDbEntry())->SetTableExists();

```

If the transaction is successful, the table has been created in the database system and is ready to be filled. Filling the existing table is done using the writer template `FairDbWriter` defined as data-member `fParam_Writer` in the parameter class.

In order to activate the writer template, the user provide a set of arguments to the template `Activate()` method which correspond to the condition associated to the data.

Validity interval: determines the interval of validity of the data to be written (see section 5.3.1)

Composition: integer number describing whether the data has a single structure (-1) or is composite $0..p-1$ with p being the number of other objects that compose this parameter object.

Version: integer number used to distinguish between multiple copies of the same (parameter,condition) duplet i.e typically needed for multiple improvements of the same calibration constants.

Database Entry: defines which entry in the database pool is used for the data storage.

LogTitle: comments stored for any update of the parameter data. It adds additional information as the comments used in code management system. For instance when using `svn -m "comment on code update"`.

The first argument i.e the *Interval of Validity* encapsulated in the `ValInterval` object, has to be provided by the parameter class itself. For this the user needs to overload the virtual function

```

// Validity frame definition
virtual ValCondition GetContext(UInt_t rid) {
return ValCondition( Detector::kGfi,
                    DataType::kData,
                    ValTimeStamp(rid));
}

```

Creating the condition as explained in section 5.3.1, the corresponding `ValInterval` object is then automatically created by calling the generic function `GetValInterval(Int_t rid)`. The `int rid` argument corresponds to the run number and is forwarded to the parameter object by the SQL I/O interface.

The other argument can be set using the static parameter factory `FairDbTutContFact` which centrally instantiates the parameter class and forwards the object pointer to the runtime database:

```
if (strcmp(name,"TUTParDefault")==0)
p=new FairDbTutPar(c->getConcatName().Data(),c->GetTitle(),c->getContext());
// Set Arguments needed for SQL versioning management
p->SetVersion(0);
p->SetComboNo(-1);
p->SetDbEntry(0);
p->SetLogTitle(name);
```

The following code fragment shows how in the `FairDbTutPar` example the writer template `Activate()` function is used:

```
// Writer Meta-Class Instance
fParam_Writer = GetParamWriter();

// Writer Activate() arguments list
// <Arguments>      <Type>      <Comments>
// Validity Interval, ValInterval
// Composition ,    Int_t      set via cont.  factory
// Version ,        Int_t      set via cont.  factory
// DbEntry ,        Int_t      set via cont.  factory
// LogTitle ,       std::string set via cont.  factory

// Activate Writer Meta-Class with the proper
// Validity Time Interval (run_id encapsulated)
fParam_Writer->Activate( GetValInterval(rid),GetComboNo(),
                        GetVersion(),GetDbEntry(),GetLogTitle());

// Object to Table mapping
*fParam_Writer<< (*this);

// Check for eventual IO problems
if ( !fParam_Writer->Close() ) {
fail = true;
cout << "-E- FairDbTutPar::Store() ***** Cannot do IO on class: " <<
GetName() << endl;
}
// end of Store()
}
```

As explained in section 5.3.1 it is only by issuing the `Bool_t Close()` that the table in the database system will be filled with updated values and the I/O completed. The functions return a `Bool_t` variable which needs to be tested in order to find out if the filling transaction

was successful (`kTRUE`) or not (`kFALSE`).

5.4.5 Accessing Condition Data

This section shows how to combine the reader template `FairDbWriter` services with the runtime database I/O interface in order to access condition data.

In section 5.2.2, we have seen that the `FairRuntimeDB::initContainers()` method calls in turn the `FairDetParTSQLIo::init(FairParSet*)` function which forwards the run number information to the concrete implementation of the overloaded

```
FairParSet::fill(Int_t runId)
```

Consequently, the user has to implement the `FairParSet::fill(Int_t runId)` using the reader template `FairDbReader`. The following code fragment shows how the implementation of the `FairDbTutPar::fill()` method:

```
void FairDbTutPar::fill(UInt_t rid)
{
    // Get Reader Meta Class
    fParam_Reader=GetParamReader();

    // Define a Condition
    ValTimeStamp ts(rid);
    ValCondition cond(Detector::kGfi,DataType::kData,ts);

    // Activate reading for this Condition
    fParam_Reader->Activate(cond, GetVersion());

    // By default use the latest row entry
    // (Other rows would correspond to outdated versions)
    Int_t numRows = fParam_Reader->GetNumRows();
    if ( numRows > 1 ) { numRows = 1; }

    for (int i = 0; i < numRows; ++i) {
        FairDbTutPar* cgd = (FairDbTutPar*) fParam_Reader->GetRow(i);
        if (!cgd) continue;
        fTopPitch = cgd->GetTopPitch();
        fTopAnchor = cgd->GetTopAnchor();
        fTopNrFE = cgd->GetNrTopFE();
        fFeType = cgd->GetFeType();
    }
}
```

The run number forwarded by the runtime database I/O interface is converted to valid timestamp using the `ValTimeStamp` services.

Additionally, in the code above only the last row stored in the reader template cache is selected. It corresponds to the usual case when the user wants to access only the last updated row in the database table which corresponds to the last update of the parameter data. The `FairDbTutPar` standard *getters* methods are then used to override the object data member values.

5.4.6 Inactive Cache

When using reader or writer I/O template, an internal caching mechanism is constantly operating in order to improve I/O performance.

Normally and as far as I/O performance is concerned, caching mechanism should never be switched off. Nevertheless in some cases it can be useful to set the caching mechanism inactive.

Let suppose the user creates a transient table as described in section 4.2.3 which replaces an existing permanent table in the database. In order to ensure that future queries done on the newly created transient table will not use data out of the cache, it is necessary to disconnect the internal cache mechanism. The following code fragment shows how to do it for the writer template `FairDbWriter`:

```
// Intanciate a Writer Template
FairDbWriter<FairDbTutPar> myWriter;
// Activate I/O for an Interval of Validiy (IOV)
myWriter.Activate(ValInterval(ioV));
// Set Cache Object Static
myWriter.TableInterface()->SetStatic();
```

The same method can be used for the `FairDbReader`.

Note that the virtual database design does not allow the cache object `FairDbCache` to be accessed directly from both I/O templates, but only via the main table manager class `FairDbTableInterface`. Indeed within the virtual database context each table is handled by a `FairDbTableInterface` that owns the associated table cache.

Using the `FairDbTableInterface::SetStatic()` method does not actually erase the data in the cache. Indeed the cached may already be in use and should not be erased until all its clients have been disconnected.

Instead the `FairDbTableInterface::SetStatic()` method marks all previously filled cache data as static, which in order words means that this data will be just ignored for all future queries.

In the code fragment above only the cache for the table associated with the parameter class `FairDbTutPar` is set static. The entire cache data associated to all other tables is untouched and still active for future queries.

5.4.7 Steering macros

This section shows how to use the virtual database and the `FairRoot` runtime database services from the ROOT (CINT) interpreter. Steering ROOT macros examples for the parameter classes described above are available in the dedicated `macros` at the location:

```
$(VMCWORKDIR)/example/tutorial5/
```

The following picture shows a snapshot of the `tutorial5`. Note that all the parameter class source files can be found in the `src` subdirectory. There are three different steering macros examples.

The first example macros are divided into *write* and *read* macros i.e the `sql_params_write.C` and the `sql_params_read.C` respectively.

This first set of macros shows how to deal with ambiguous query i.e multiple version of the same parameter data within the virtual database context as explained in details in section 5.4.4.

```

drwxr-xr-x  6 denis  staff  2048 Oct  2 09:49 ./
drwxr-xr-x 14 denis  staff  4768 Oct  2 15:59 ../
drwxr-xr-x  8 denis  staff  2728 Oct  2 09:49 .svn/
-rw-r--r--  1 denis  staff  1668 Oct  2 09:49 CMakeLists.txt
drwxr-xr-x 17 denis  staff  5788 Nov 21 10:10 macros/
drwxr-xr-x 13 denis  staff  4428 Nov 22 13:33 src/

```

Two different versions of the same parameter data having different context names ¹² `TutParDefault` and `TutParAlternative` are first read from a text-based file first input with the runtime database Ascii I/O interface `FairParAsciiFileIo`:

```

Int_t sql_params_write()
{
// Generate a unique RunID
FairRunIdGenerator runID;
UInt_t runId = runID.generateId();

// Create the Runtime Database (parameter manager class)
FairRuntimeDb* db = FairRuntimeDb::instance();

// Create in memory the relevant container
FairDbTutPar* p1 = (FairDbTutPar*)(db->getContainer("TUTParDefault"));
FairDbTutPar* p2 = (FairDbTutPar*)(db->getContainer("TUTParAlternative"));

// Set the Ascii IO as first input
FairParAsciiFileIo* inp1 = new FairParAsciiFileIo();

TString work = getenv("VMCWORKDIR");
TString filename = work + "/example/Tutorial5/macros/ascii-example.par";

inp1->open(filename.Data(),"in");

// Set the ASCII interface as
// first input
db->setFirstInput(inp1);

```

The SQL I/O interface is used as a second input of the runtime database manager class `FairRuntimeDb` for storing the parameter object in a database system:

¹² The parameter have different *context name* in the runtime database terminology. This context should not be confused by the *Condition* and the *Interval Of Validaty* defintion within the virtual database context.


```

// Set the SQL based IO as second input
FairParTSQLIo* inp2 = new FairParTSQLIo();
inp2->SetVerbosity(1);
// Shutdown Mode ( True, False )
inp2->SetShutdown(kTRUE);
// Activate the SQL I/O interface
inp2->open();

// Set The TSQL I/O as
// second input
db->setSecondInput(inp2);

// Runtime Database <INIT>
// containers are intialised from Ascii input
// with assigned RunId

db->initContainers(runId);

cout << endl;
cout << "-I- Initialization from Ascii File done using RunID:" << runId <<
endl;
cout << endl;

p1->Print();
p2->Print();

```

The actual SQL I/O including table creation and object to table mapping is done when calling the method `FairRuntimeDb::writeContainers()` from the main parameter manager singleton:

```

// <WRITE> back containers to the user-defined
// Database using the Sql based IO of the
// second input.

// Data Storing is done in the database
db->setOutput(inp2);
// SQL I/O triggered
db->writeContainers();

cout << endl;
cout << "-I- Parameters succesfully written to DB with RunID:" << runId <<
endl;
cout << endl;

if (db) delete db;
return 0;
}

```

The corresponding *read* macros initializes the previously written parameter data from the

database by defining as first input the SQL I/O interface. Parameter data is initialized using a user-defined timestamp converted into a run number for the runtime database. For the same timestamp i.e same run number, the two versions `TUTParDefault` and `TUTParAlternative` of the same parameter class are filled properly.

```

Int_t sql_params_read()
{
// Create a Runtime Database singleton.
FairRuntimeDb* db = FairRuntimeDb::instance();

// Set the SQL IO as first input
FairParTSQLIo* inp = new FairParTSQLIo();
// Verbosity level
inp->SetVerbosity(1);
inp->open();
db->setFirstInput(inp);

// Create the container via the factory if not already created
FairDbTutPar* p1 = (FairDbTutPar*)(db->getContainer("TUTParDefault"));
FairDbTutPar* p2 = (FairDbTutPar*)(db->getContainer("TUTParAlternative"));

// Create a dummy runID using date in UTC from which
// corresponding parameters will be initialized
ValTimeStamp tStamp(2013,10,22,15,00,00);
UInt_t runId = tStamp.GetSec();
cout << "-I- looking for parameters at runID " << runId << endl;
cout << "-I- corresponding time in runID (UTC) " << tStamp.Format("iso") <<
endl;

// Use the generated RunID to initialized the parameter
// using the SQL-based IO input
db->initContainers(runId);
// Get the container after initialization
// from the RuntimeDB
FairDbTutPar* pp1 = (FairDbTutPar*)(db->getContainer("TUTParDefault"));
FairDbTutPar* pp2 = (FairDbTutPar*)(db->getContainer("TUTParAlternative"));

cout << "-I- Reading Parameter data from SQL Database:" << endl;

pp1->Print();
pp2->Print();

if (db) delete db;
return 0;
}

```

In the same `macros` directory, similar examples can be found dealing with storing and accessing parameter data using more complex data types as data members i.e `sql_param_write_bin.C` and `sql_param_write_bin.C` respectively. The complex data types are represented as binary string column in the corresponding relational table.

Additionally a simple example showing the usage of the SQL I/O interface within a `FairTask` example is available. In the `FairTask::SetParContainer()`, `FairTask::Init()` and `FairTask::ReInit()` one uses the SQL I/O interface `FairParTSQLIo` in a similar way as any other I/O interface supported by the runtime database.

All corresponding source codes and steering macros can also be found in the `src` and `macros` directory respectively. The steering macros using `FairTask` are

```
$(VMCWORKDIR)/example/tutorial5/macros/runsim.C  
$(VMCWORKDIR)/example/tutorial5/macros/runana.C
```

and the source code of the simple `FairTask` implementation is found at:

```
$(VMCWORKDIR)/example/tutorial5/src/FairDbTutAccessRtdbTask.h  
$(VMCWORKDIR)/example/tutorial5/src/FairDbTutAccessRtdbTask.cxx
```

Bibliography

- [1] Coordinated universal time: http://de.wikipedia.org/wiki/Koordinierte_Weltzeit. (Cited on page 17.)
- [2] Standard template library <http://www.cplusplus.com/reference/stl>. (Cited on page 22.)
- [3] <http://www.oracle.com/us/products/database/berkeley-db/index.html>. *Oracle Berkeley DB Products*, 2010. (Cited on page 9.)
- [4] <http://web-docs.gsi.de/halo/docs/hydra/classDocumentation/root5.22/hydra.pdf>. *Hydra2 Manual*, February 2, 2011. (Cited on page 9.)
- [5] Eric Brewer. Towards robust distributed systems. (keynote at the acm symposium on principles of distributed computing - podc). *Portland, Oregon*, July 200. (Cited on pages 7 and 8.)
- [6] R. Brun and F. Rademakers. Root - an object oriented data analysis framework. *Nucl. Instr. and Meth. in Physics Research A*, A389:81–86, 1997. (Cited on page 3.)
- [7] Large Hadron Collider. <http://home.web.cern.ch>. (Cited on page 6.)
- [8] MySQL community. <http://dev.mysql.com>. (Cited on page 3.)
- [9] PostgreSQL community. <http://www.postgresql.org>. (Cited on page 3.)
- [10] Apache CouchDB. <http://couchdb.apache.org>. (Cited on page 5.)
- [11] C. J. Date and Hugh Darwen. A guide to the sql standard : a users guide to the standard database language sql, 4th ed. *Addison Wesley*, ISBN 978-0-201-96426-4, 1997. (Cited on page 5.)
- [12] Mongo DB. <http://www.mongodb.org>. (Cited on page 5.)
- [13] Kristina Chodorow et al Eliot Horowitz, Dwight Merriman. Mongoddb manual - mapreduce (<http://www.mongodb.org/display/docs/mapreduce>). *Wiki article, version 80*, November 2010. (Cited on page 8.)
- [14] Fay Chang et al. Bigtable: A distributed storage sytem for structured data. *OSDI*, Google Inc., 2006. (Cited on page 9.)
- [15] ATLAS Experiment. <http://atlas.web.cern.ch>. (Cited on page 5.)
- [16] CMS Experiment. <http://cms.cern.ch>. (Cited on page 6.)
- [17] LHCb Experiment. <http://lhcb.web.cern.ch>. (Cited on page 6.)
- [18] I.Hrinacova F. Carminati, R. Brun and A.Morsch. Virtual monte carlo. In *Computing in High Energy and Nuclear Physics*, pages 24–28, 2003. (Cited on page 10.)
- [19] FairRoot. <http://fairroot.gsi.de>. (Cited on page 3.)
- [20] Apache Hadoop. <http://hadoop.apache.org>. 41. (Cited on page 5.)
- [21] IBM-DB2. <http://www.ibm.com/software/data/db2>. (Cited on page 5.)

- [22] Bob Ippoliti. Drop acid and think about data <http://blip.tv/file/1949416>. *Talk at Pycon*, March 2009. (Cited on page 7.)
- [23] Oracle. <http://www.oracle.com>. (Cited on page 3.)
- [24] Carlo Strozzi. Nosql a relational database management system. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, 2007-2010. (Cited on page 5.)

FairRoot Virtual Database

Abstract: Keywords: FairRoot, Condition Database , SQL, NoSQL
