École Polytechnique Fédérale de Lausanne (EPFL)
Faculté Informatique et Communications (I&C)

Semester Project Report

Metadata Front-end for Shore-MT Storage Manager

Student: Bao Duy TRAN (210215)

Supervisor: Prof Anastasia Ailamaki
Advisers: Pınar Tözün & Danica Porobic

September 2012 – January 2013

Abstract

Shore-MT is a scalable storage manager offering high performance on multi-core architectures [1]. This report documents the design and implementation of Shore-MT front-end, a dynamic relational application layer for Shore-MT with metadata management capability. The work was conducted as an EPFL semester project.

1. Introduction

1.1. Original SHORE

Scalable Heterogeneous Object REpository (**SHORE**) [2] was a project initiated at the University of Wisconsin-Madison (US) in the 1990s, aiming to combine object-oriented database and file system technologies into a single object persistent system. The UNIX-based system supports single-core CPUs and was built upon a layered architecture consisting of a storage manager, a value-added server and a data language.

One major component of the project, **SHORE storage manager**, has since played a vital role in various published database studies [3]. In particular, it has been employed in a number of research-oriented special-purpose database management systems (DBMSs) like TIMBER [4], Paradise [5] and λ-DB [6].

1.2. Shore-MT storage manager

With the advent of modern multi-core architectures and the growing need for scalability, **Shore-MT storage manager** [1] was developed in the late 2000s by the Data-Intensive Applications and Systems (DIAS) Laboratory at EPFL (Switzerland) [7] as a multi-threaded port of the original SHORE storage manager. Since its release, Shore-MT has been used in the academia as a highly scalable open-source solution for persistent data management in multi-threaded applications.

Shore-MT storage manager is essentially a C++ system library. Just as SHORE, application layers can be built on top of Shore-MT in the form of value-added servers. One such example is **Shore-Kits** [8], an experimental framework which implements several industry-standard Transaction Processing Performance Council (TPC) benchmarks [9]. Shore-Kits are used at DIAS for performance evaluations of Shore-MT features.

1.3. Motivations for a dynamic front-end

Shore-MT offers low-level application programming interface (API) access to a high-performance persistent storage manager with restrictions on neither interaction patterns nor data models. This implies flexibility for application developers. In other words, the application layer might as well be a traditional DBMS based on the relational model [10] or other alternatives.

Nevertheless, there is no denying that the relational model remains the most pervasive in practice. Despite certain drawbacks, many enterprise applications and academic research activities still focus on this ubiquitous data model. Flexible as it might be in general, Shore-MT could be cumbersome in the realm of relational data management. All activities (e.g. creating a database, defining a schema, dropping a table, etc.) have to be hard-coded against Shore-MT C++ API and repeated from one application to another. E.g. Shore-Kits [8] are based on the relational model and suffer from this problem.

Thus there is a motivation for a convenient, generic and reusable front-end which encapsulates commonly-used relational features, especially data definitions and administrative tasks. Such a system shall act as a dynamic utility layer above Shore-MT, providing applications with facilities for relational metadata management.

Furthermore, in the context of academic research on performance-critical databases, quick prototyping from a scripting environment (such as Python [11]) is usually conducted before transaction processing code is formulated. This inevitably requires the relational front-end to expose crucial features via a scripting interface besides traditional C++ API. In the meantime, there exists libraries to assist the interfacing process, such as the Simplified Wrapper and Interface Generator (SWIG) [12].

All in all, there are strong motivating factors for a Shore-MT metadata front-end with scripting support, which is the topic of this semester project.

1.4. Project objectives

The general objectives identified at the outset of the project were as follows:

- To familiarise with Shore-MT concepts and API using existing documentation [13–15] and Shore-Kits implementation [8] as references.
- To investigate SWIG [12], a cross-language interfacing library, and identify the suitability of direct interfacing of Shore-MT API to a scripting environment.
- To design and develop a relational metadata front-end for Shore-MT which supports persistent and reflective schemas, either natively in C++ then interfaced to a scripting environment, or on top of an already-interfaced Shore-MT API.
- To implement interactive console applications which allow simple relational database management activities, mainly to demonstrate the front-end's functionalities.
- To ensure that all deliverables are as transferable, maintainable and reusable as possible, and that all features and design decisions are documented.

1.5. Project scope

- Languages inherently differ in concepts, syntax and memory models. Despite SWIG's multi-language support, fine-tuning is inevitable for a specific target. Hence, Python [11] shall be prioritised as the target environment of choice.
- The following relational database operations shall be supported by the front-end:
  - Initialisation and shut-down of the underlying Shore-MT storage manager;
  - Creation and deletion of relational databases;
  - Selection of relational databases (for use in subsequent operations);
  - Creation (given a schema) and deletion of relational tables;
  - Insertion of tuples into relational tables;
  - Retrieval and display of whole relational tables (i.e. all tuples);
- Parsers for data definition, manipulation and control languages (e.g. a Structured Query Language (SQL) [16] parser) are not required. The front-end shall instead present proper C++ and scripting APIs while the interactive console application can accept user inputs in non-standard formats.
- Functionalities shall remain the sole focus of this inaugural Shore-MT front-end project. In other words, no performance optimisations are required.

1.6. Project schedule

The project officially lasted 14 weeks during the autumn semester of academic year 2012/13 (18 Sept – 21 Dec 2012), with approximately 18 working hours per week. Nonetheless, preliminary preparation had started in early Sept 2012 and wrap-up tasks extend till Jan 2013. An actual breakdown of main tasks is reported below:

1. Familiarisation with Shore-MT through documentation, sample codes, Shore-Kits implementation and self-written tests (*early – late Sept 2012*).
2. Investigation of SWIG capabilities and usage; Feasibility study of SWIG interfacing with Shore-MT API (*late Sept – late Oct 2012*).
3. Design and implementation of Shore-MT front-end (*late Oct – mid-Dec 2012*).
4. Design and implementation of demo applications (*mid – late Dec 2012*).
5. SWIG interfacing with the Python scripting environment (*late Dec 2012*).
6. Report writing, handover of deliverables and presentation (*early Jan 2013*).

1.7. Organisation of the report

The rest of the report is organised as follows: Section 2 reviews Shore-MT concepts relevant to the project. Section 3 presents important design decisions and an architectural overview of Shore-MT front-end. Section 4 delves into the details of individual modules while a separate discussion is dedicated to the Python-interfacing task in Section 5. The report concludes in Section 6 with recommendations for future work.

2. Shore-MT concepts

While Shore-MT offers a wide range of capabilities in terms of storage structures, buffer pool management, system recovery, transaction management, etc. [17], only relevant concepts are reviewed below. Most fundamental elements are inherited from SHORE and the reader should refer to the original documentation [13–15] for further details.

2.1. Storage structures

In essence, Shore-MT persists data onto raw storage **devices** which can either be physical or logical. Hard disk drives are typical examples of physical devices whereas any raw files in a UNIX file system can act as logical devices. Devices are identified by their UNIX file system paths and need to be formatted by Shore-MT before use.

By design, there can be multiple **volumes** on a formatted device; however, there is only a single volume in practice, according to the latest version of Shore-MT. Storage **files** are the most basic structures found on Shore-MT volumes. Each file consists of multiple **records** where user data are stored. Every record has an optional header and a body, both of which are raw binary data. Volumes, files and records are identified by volume, file and record IDs, respectively.

In order to locate data more efficiently, application can build **indices** (identified by index IDs) which associate keys (typically a portion of the record) to values (either the record ID or any arbitrary data). B+ tree [18] and R* tree [19] are amongst the index types supported. Indices and files are collectively termed '**stores**'; both can be scanned sequentially while look-ups by keys are only possible with indices. Besides application-specific indices, every volume has a well-known **root index** for bootstrap purpose.

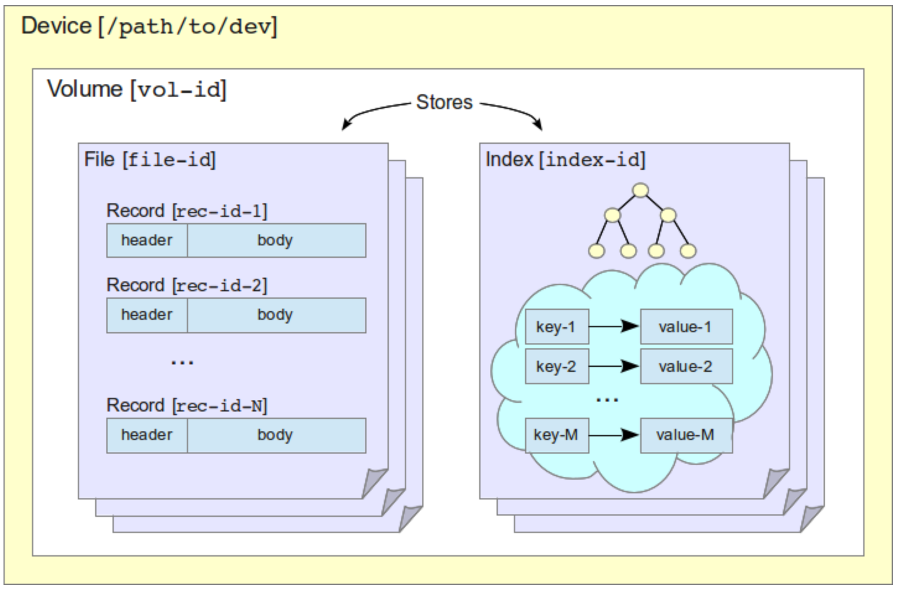


Figure 1: Logical storage structures on a Shore-MT device

A logical view of Shore-MT storage structures is depicted in Figure 1. It is noteworthy that what data to be stored in Shore-MT files and how these data are formatted are up to the application layer. Similarly, contents of indices, including volume root indices, are outside the purview of Shore-MT.

2.2. Initialisation, threading and transactions

Shore-MT can be **initialised** by instantiating the `ss_m` class with appropriate configuration options. It follows that Shore-MT can be shut down by destroying this `ss_m` instance. Although the library is written in C++, the main API is accessible via static methods of the `ss_m` class. This greatly simplifies implementations of the application layer.

Shore-MT requires that all API invocations be conducted on specialised **threads** derived from `smthread_t`, which is in turn an extension of POSIX thread. Failure to comply to this requirement results in errors being reported or drastic segmentation faults. Subthreads can be spawned from existing ones but all must be derived from `smthread_t`.

Native support for **transaction** management is provided by Shore-MT. Certain operations must be executed in the context of a transaction, while others (mostly administrative tasks) are not allowed to appear in transactions.

3. Front-end design & implementation

3.1. Preliminary design decision

According to the project objectives (Section 1.4), two approaches could initially be suggested for the design and implementation of Shore-MT front-end:

1. Interface Shore-MT API to Python, then build Shore-MT front-end as a scripting module on top of this wrapped API.
2. Build core components of Shore-MT front-end natively in C++ against Shore-MT API, then interface the resultant module to Python.

These two approaches differ only in the level where interfacing to Python occurs. After rigorous examination of Shore-MT, thorough investigation of SWIG and careful prototyping, the **2nd approach** was adopted, for the following main reasons:

- *Incompatibility & complexity*: Shore-MT was not developed with cross-language compatibility at its core. The API sometimes requires parameters of complex types and employs advanced C/C++ features absent in other languages.
- *Efforts not paying off*: Assuming that, despite issues presented above, Shore-MT could be available as a Python API, the application layer must still interact in the same manner. In other words, the fact that the application can be written in a higher-level language adds little value, as the developer still needs to be aware of the C++ nature and the resultant codes potentially look no much simpler.
- *Threading hindrance*: Shore-MT has special threading requirements (Section 2.2). Dynamic applications therefore need a threading strategy that satisfies these requirements. Implementation of such a strategy is safer and more elegant with low-level access to underlying threading libraries.

3.2. Architectural overview

Functional design of Shore-MT front-end prioritises modularisation (separation of concerns with clear dependencies) and attempts to identify generic reusable components.

With this philosophy, an overall architecture was formulated and refined as the project progressed. The finalised architecture is presented in Figure 2, which shows module *types* (primary, supportive or executable), *roles* (external, peripheral or internal), *natures* (native, scripting or hybrid) and *inter-dependencies* amongst modules.

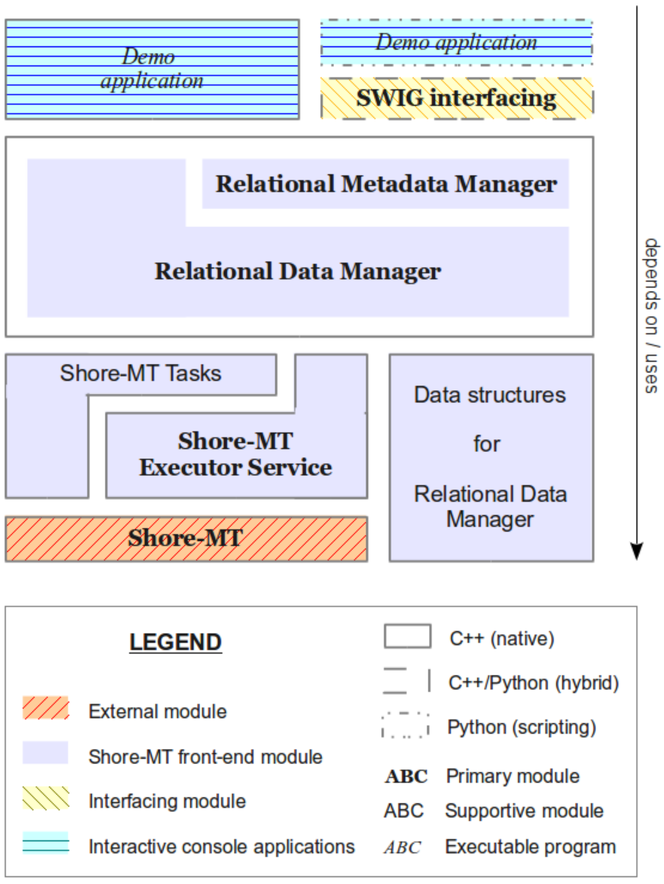


Figure 2: Architecture of Shore-MT front-end

It is worth mentioning that not all boxes in Figure 2 represent physical modules as some denote logical separation of responsibilities. At a glance, the following can be noted:

- Only the Executor Service module and associated task implementations are dependent on Shore-MT; data structures for the Relational Data Manager are not.
- Interactions with Shore-MT API are formulated as tasks and routed to Shore-MT via an Executor Service (see Section 4.1).
- The Relational Data Manager (persisting and managing user data) is employed reflectively to realise a Metadata Manager (persisting and managing metadata).
- Native and scripting applications are structured similarly, except that the latter requires an intermediate SWIG layer comprising C++ and Python glue codes.

3.3. Object-oriented approach

Design and implementation of Shore-MT front-end follow the object-oriented paradigm [20], in order that deliverables remain well-structured and maintainable as complexity grows. Despite being debatable whether this is the best, it is still better than not adopting any standardised principles. In Section 4, object designs of individual modules will be shown in Unified Modelling Language (UML) [21] class diagrams.

3.4. C++ implementation

Shore-MT front-end[1] was developed as a C++ project on UNIX-like environment. Just like Shore-MT and many open-source packages, the project employs GNU Autotools **build system** [22–24] with GNU Make [25] and GNU Compiler Collection (GCC) [26]. It is **revision-tracked** with Mercurial (HG) distributed source control management [27]. The reader is advised to consult Appendix A.1 for set-up and build instructions.

Shore-MT front-end has been implemented and verified on server `diassrv2.epfl.ch` of DIAS Laboratory, linking against HG changeset `268:729613cf211e` of Shore-MT cloned from `ssh://diasgate.epfl.ch/HG/shore-storage-manager`. Most importantly, it **depends** solely on Shore-MT installation[2] and C++ Standard Libraries [28].

The project makes use of logical **file naming** conventions as well as manageable grouping of C++ classes into source files (see Appendix B). Consistent **coding style** has been adopted together with comprehensive **documentation** in the forms of in-line comments and Doxygen [29] for classes, attributes and methods. This makes the project comparable with Shore-MT in terms of readability, maintainability and transferability.

4. Individual front-end modules

4.1. Executor Service – A generic Shore-MT utility

Owing to Shore-MT's threading particularities (Section 2.2) and the front-end's dynamic nature (pattern of invocations determined at run-time), a unified threading scheme is necessary to formalise Shore-MT API invocations as task executions via one or many authorised threads. This is to avoid unmanaged spawning of threads which is cumbersome in development and risky in execution. The Executor Service module is the solution to this issue, constituting a major side contribution of the project.

[1]All modules in Figure 2 except the scripting application and its interfacing layer (see Section 5).
[2]Artefacts of `make install`, consisting of header files and binary objects (not the source project).

4.1.1. Functional design

Executor Service is a generic and reusable Shore-MT utility. Bearing zero dependency on the front-end under development (see Figure 2), the module depends on Shore-MT and ease its use under threading restrictions. While its capabilities remain limited to those needed by the front-end, it is extensible to a comprehensive library.

Executor Service encapsulates a private **worker thread** (derived from `smthread_t`) while exposing a **control API** to be invoked on the user thread. In a typical usage scenario (see Figure 3), the user submits a **task** (user-defined execution content) to the Executor Service which accepts and enqueues it to an internally-managed **task queue**. Processing is conducted by the worker thread which normally waits as long as the queue is empty. Otherwise, tasks are dequeued and executed in a first-in-first-out (**FIFO**) manner.

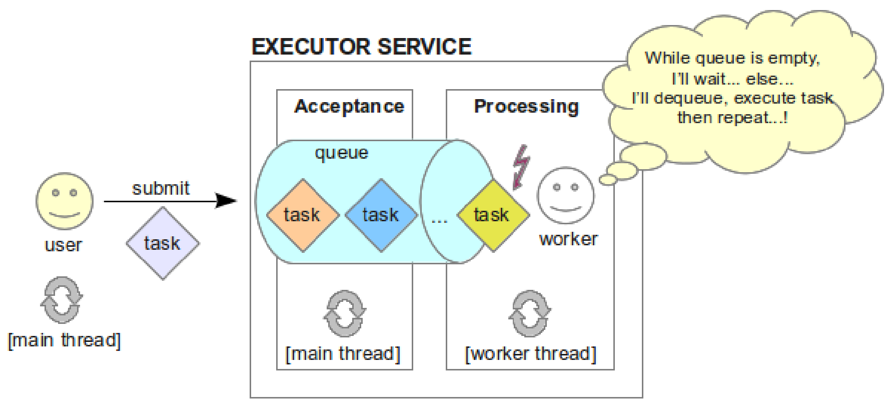


Figure 3: Usage scenario of Shore-MT Executor Service

There are two modes of executions: In **synchronous** mode, the user thread is blocked from task submission till completion. In **asynchronous** mode, on the other hand, the user thread proceeds and completion of task execution can be notified via a simplified observer pattern[3] [20].

Proper **signalling scheme** has to be put in place to unblock (1) the worker thread upon task availability and (2) the user thread upon task completion in synchronous mode. Synchronisations are also required to protect data structures shared amongst threads.

4.1.2. Object design

Object design of Shore-MT Executor Service is presented in Figure 4. Typically, user interacts with a `ShoreExecutorService` instance and submits tasks (instances of some concrete implementations of `ShoreTask`) by invoking either `syncExec()` or `asyncExec()`, depending on the desired execution mode.

[3]Also called 'publish-and-subscribe', or colloquially 'listener' pattern.

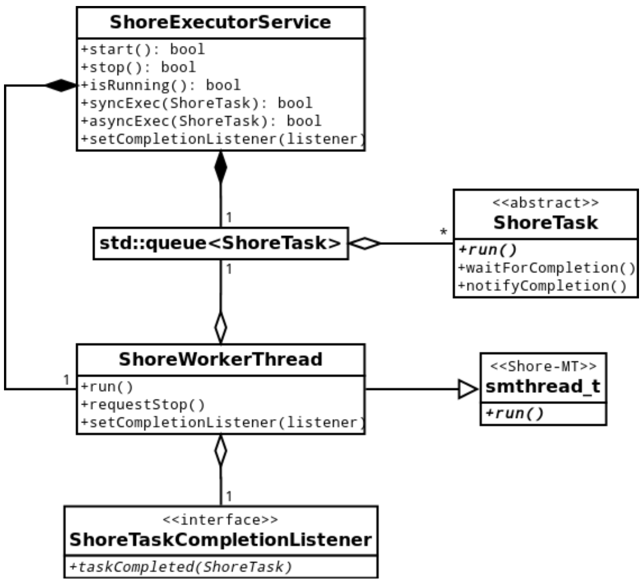


Figure 4: UML class diagram of Shore-MT Executor Service module

4.1.3. Additional remarks

Shore-MT Executor Service is a special case of thread-pool executor[4] with a single internal thread operating off an unbounded task queue. The reason calling for Shore-MT Executor Service is that public libraries are based on standard threads while Shore-MT authorises *only* `smthread_t`-derived threads.

Also, similar scheme has been implemented in Shore-Kits [8]. Unfortunately, this legacy resource is not generic and modular enough for reuse in Shore-MT front-end. Specifically, the inheritance hierarchy appears to have been mixed with Shore-Kits' specific features and no generic task definitions are possible.

4.1.4. Synchronous execution example

This Section demonstrates the use of Executor Service and tasks to *synchronously* access Shore-MT API, without having to explicitly deal with threads. Assume we want to implement a hypothetically simple algorithm which consists of executing two Shore-MT API methods sequentially and does not care about return statuses:

```
void myShoreMtOperation() {
    ss_m::doSomething();
    ss_m::doSomethingElse();
}
```

[4]For instance, the equivalent exists in standard Java [30] and Boost-based C++ libraries [31].

According to Shore-MT threading requirements (Section 2.2), the above is only authorised to run on an `smthread_t`-derived thread and we may want to implement our own threading scheme to achieve that. However, if there is a running `ShoreExecutorService` instance available, we can instead do the following:

```
void myShoreMtOperation(ShoreExecutorService* myExecSvc) {
    DoTwoThingsTask myTask;
    myExecSvc->syncExec(&myTask);
}
```

where `DoTwoThingsTask` is `ShoreTask`-derived and its `run()` method is defined as:

```
void DoTwoThingsTask::run() {
    ss_m::doSomething();
    ss_m::doSomethingElse();
}
```

Now `myShoreMtOperation()` can be invoked on *any* thread without bothering about Shore-MT threading requirements. Furthermore, it will appear as if the execution were on the caller's thread, thanks to the synchronous mode.

4.2. Front-end tasks & threading

In Shore-MT front-end, specific Executor Service tasks are defined by supplying implementations of `ShoreTask`. Unlike the dummy sample sketched in Section 4.1.4, their `run()` methods access Shore-MT API to perform meaningful units of work required by the front-end. In addition, these units of work are general enough to facilitate internal reuse. Figure 5 describes the inheritance hierarchy of various front-end tasks.

As can be seen from Figure 5, all tasks are derived from `BaseTask`, which augments error reporting capabilities to `ShoreTask`. They can be broadly categorised as follows:

- *Administrations*: Initialise/Shut down Shore-MT: `CreateSmTask`, `DestroySmTask`.
- *Transaction controls*: `BeginTransactionTask`, `EndTransactionTask`.
- *Manipulations of a device* (at a given path):
  - Mount a device (and retrieve the single volume ID): `MountDevTask`.
  - Initialise a device (i.e. format the device, create a single volume on it and retrieve the volume ID): `InitDevTask`.
  - Destroy a device (i.e. destroy all volumes): `DestroyDevTask`.
- *File manipulations*:
  - Create a file in a volume given the volume ID: `CreateFileTask`.
  - Destroy a file identified by an ID: `DestroyFileTask`.
  - Append an empty-header record to a file identified by an ID, using the given raw binary data as record body: `CreateRecordTask`.

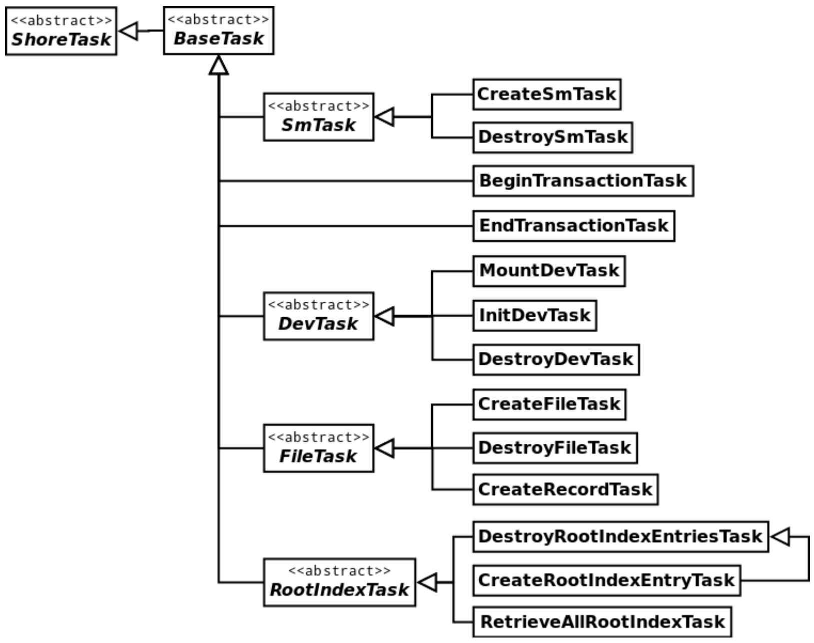


Figure 5: UML class diagram of specific Executor Service tasks

- *Root index manipulations of a volume* (identified by an ID):
  - Destroy all entries having a particular key: `DestroyRootIndexEntriesTask`.
  - Append a new mapping entry key $\rightarrow$ value: `CreateRootIndexEntryTask`.
  - Scan and retrieve all entries: `RetrieveAllRootIndexTask`.

4.2.1. Threading strategy in Shore-MT front-end

In Shore-MT front-end, an Executor Service is instantiated upon top-level initialisation and passed to subordinate modules. Whenever access to Shore-MT API is required, a suitable task is constructed and submitted to the Executor Service for execution.

Consequently, all accesses to Shore-MT API are serialised on a single dedicated thread – the Executor Service's worker thread. While this facilitates all requirements of the project, with the Executor Service's extensibility, new threading schemes can be adapted.

Furthermore, despite Executor Service's flexibility, only synchronous executions are employed by the front-end. To the caller, all operations appear as if they were performed directly on her own thread, simplifying internal operations and external usage.

4.3. Relational Data Manager

While threading modules (Sections 4.1 and 4.2) provide fundamental infrastructure, Relational Data Manager is the most significant module in terms of functionalities.

4.3.1. Functional design

Basically, this is the application layer which adds relational concepts and operations to Shore-MT. It encompasses *all* relational database operations listed in Section 1.5 and can be regarded as a primitive database system based on the relational model [10].

- The system manages a collection of **databases**, each of which comprises a collection of **tables**[5]. Each table is a collection of **tuples** (ordered sequence of **field** values). All tuples in a table adhere to a pre-defined and common schema.
- A **schema** defines tuple fields and their order. It is a sequence of **field descriptions**, each of which has (1) a name, (2) a data type, and configurations like (3) size limit and (4) whether null values are permitted.
- The system allows all administrative, data-retrieval and data-manipulation **operations** mentioned in Section 1.5. At most one database is **selected** (in use) at a time. The convention is that operations work on this selected database.

There is no persistence of relational metadata. This feature is realised by the Relational Metadata Manager discussed in Section 4.4. With the Data Manager alone, the user could experience a fully-featured Shore-MT front-end but no database state would ever survive a simple application restart.

4.3.2. From relational model to storage structures

In Shore-MT front-end, relational entities are mapped to Shore-MT storage structures (cf. Section 2.1). All these mappings are visualised in Figure 6 (cf. Figure 1):

- **Database** $\leftrightarrow$ **Device**: Each relational database is stored in its entirety on a dedicated Shore-MT device, i.e. a database is self-contained on its device[6]. This is potentially beneficial for independent attachments of database devices.
- **Table** $\leftrightarrow$ **File**: Each relational table is stored in its entirety in a Shore-MT file located on the device (volume) of the respective database.
- **Tuple** $\leftrightarrow$ **Record**: Each relational tuple is stored as a record in the Shore-MT file of the respective table. The record header shall be empty while the record body shall contain the tuple's field values formatted as binary data. This disk format of relational tuples is elaborated in Section 4.3.4.

[5]User indices to facilitate data look-up are outside the scope of this project.
[6]With only single-volume devices (see Section 2.1), a database is also self-contained on a volume.

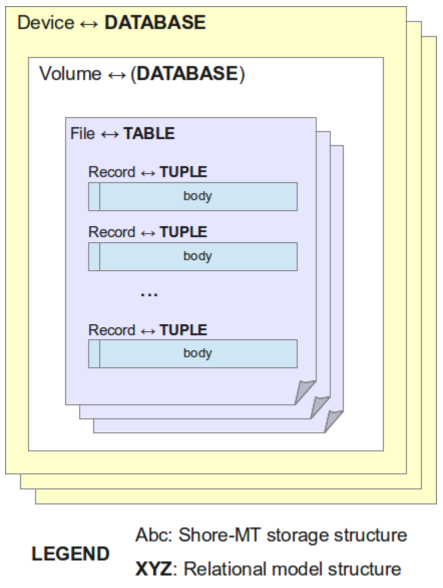


Figure 6: Relational databases in Shore-MT storage structures

4.3.3. From SQL to system data types

SQL types define types of data stored in relational tuple fields. They are specified as part of the table schema. In general, SQL data types slightly differ from one DBMS to another. In this inaugural Shore-MT front-end, only the most fundamental types are supported and they closely follow SQL standards [16]. Table 1 lists all supported SQL types in Shore-MT front-end.

Table 1: SQL data types in Shore-MT front-end

SQL type	C++ type	Description	Single unit?	Fixed length?
`BIT`	`bool`	Boolean (bit)	√	√
`SMALLINT`	`short`	Short integer	√	√
`INT`	`int`	Integer	√	√
`LONG`	`long`	Long integer	√	√
`FLOAT`	`float`	Floating-point	√	√
`DOUBLE`	`double`	Double-precision floating-point	√	√
`CHAR`	`char[]`	Fixed-length character string		√
`VARCHAR`	`char[]`	Variable-length character string		

It is noteworthy that `CHAR` and `VARCHAR` types only differ in terms of fixed- versus variable-length in the disk format of a relational tuples (see Section 4.3.4). When a schema is specified, **size configuration** of `CHAR` refers to the fixed length while that of `VARCHAR` indicates the maximum possible length. Size configuration is always given as the number of **units** (e.g. `int` and `char` are units of SQL types `INT` and `VARCHAR`, respectively), never in bytes; it is therefore irrelevant for single-unit types. **Unit size** is however given in bytes and computed as `sizeof()` of the corresponding unit.

In the C++ implementation of Shore-MT:

- SQL types are realised as an `enum` called `ShoreSqlType` and referred to by their `enum` names[7]. Due to C++ limitations[8], associated properties and functionalities have to be global functions with the SQL type concerned passed as parameter.
- SQL values are realised as a `union` called `ShoreSqlVal` where actual mappings to system types take place. Value assignments and retrievals require an indication of `ShoreSqlType`. Therefore, `ShoreSqlVal` is just a low-level, type-unsafe, and supportive data structure for classes like `ShoreField` (see Section 4.3.5).

4.3.4. From relational tuples to Shore-MT records

As mentioned in Section 4.3.2, a tuple has to be serialised into binary data before being written to a Shore-MT record. The reverse process takes place when the record is read to reconstruct the logical tuple. This Section describes the forward serialisation process. In fact, the record format has been inspired by Shore-Kits [8] implementation.

Tuple fields are first categorised as (1) nullable, (2) fixed-length and (3) variable-length while keeping their relative order intact. Nullable fields of group (1) refer to those permitting SQL `NULL` as value, no matter what SQL types. Groups (2) and (3) apply to the rest of the fields according to their SQL types. The tuple can then be formatted into four consecutive binary blocks (Figure 7):

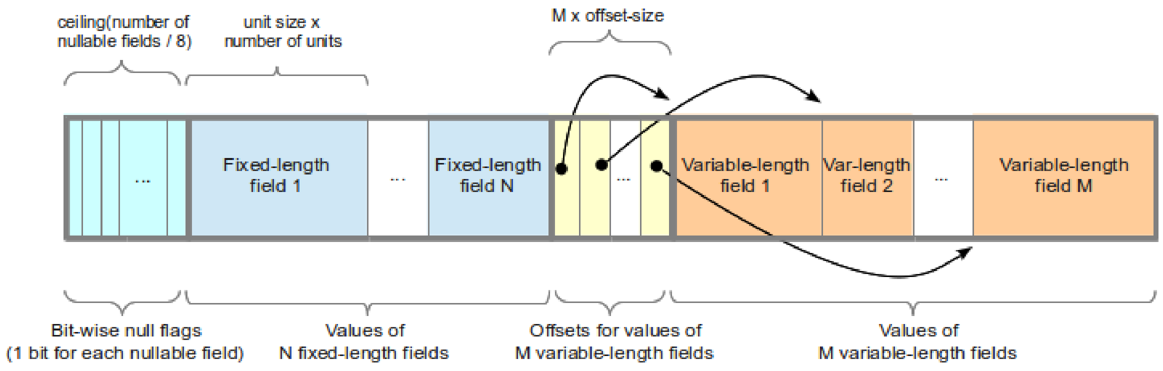

ceiling(number of nullable fields / 8)
unit size x number of units
M x offset-size
...
Fixed-length field 1
...
Fixed-length field N
...
Variable-length field 1
Var-length field 2
...
Variable-length field M
Bit-wise null flags (1 bit for each nullable field)
Values of N fixed-length fields
Offsets for values of M variable-length fields
Values of M variable-length fields

Figure 7: Relational tuple formatted as binary data in Shore-MT record body

1. **Null flags**: This block reserves one bit for each nullable field of group (1) while respecting their relative order. The bit is set if the respective field contains SQL `NULL` value. If the number of nullable fields is not a multiple of 8, the block is right-padded with dummy bits to reach the byte boundary.

[7]SQL type names in Table 1 prefixed with "`SQL_`".
[8]In C++, an `enum` is not a class with methods [32].

2. **Fixed-length fields**: Values of fixed-length fields of group (2) are appended one after another into this block. Each field can occupy different lengths depending on their SQL types; nevertheless, these are known via the schema.
3. **Variable-length field offsets**: Offsets to variable-length field values (stored in block 4) are written in this block. Offsets are computed with respect to the very beginning of the byte sequence (offset 0) and stored as `unsigned int` values.
4. **Variable-length fields**: Values of variable-length fields of group (3) are appended one after another into this block. Although their lengths cannot be determined from the schema, offsets in block 3 help locate them.

4.3.5. Relational data structures

Relational Data Manager uses several data structures to represent the relational model. These are conceptually depicted in Figure 8. No interactions with Shore-MT API are made from these data structures.

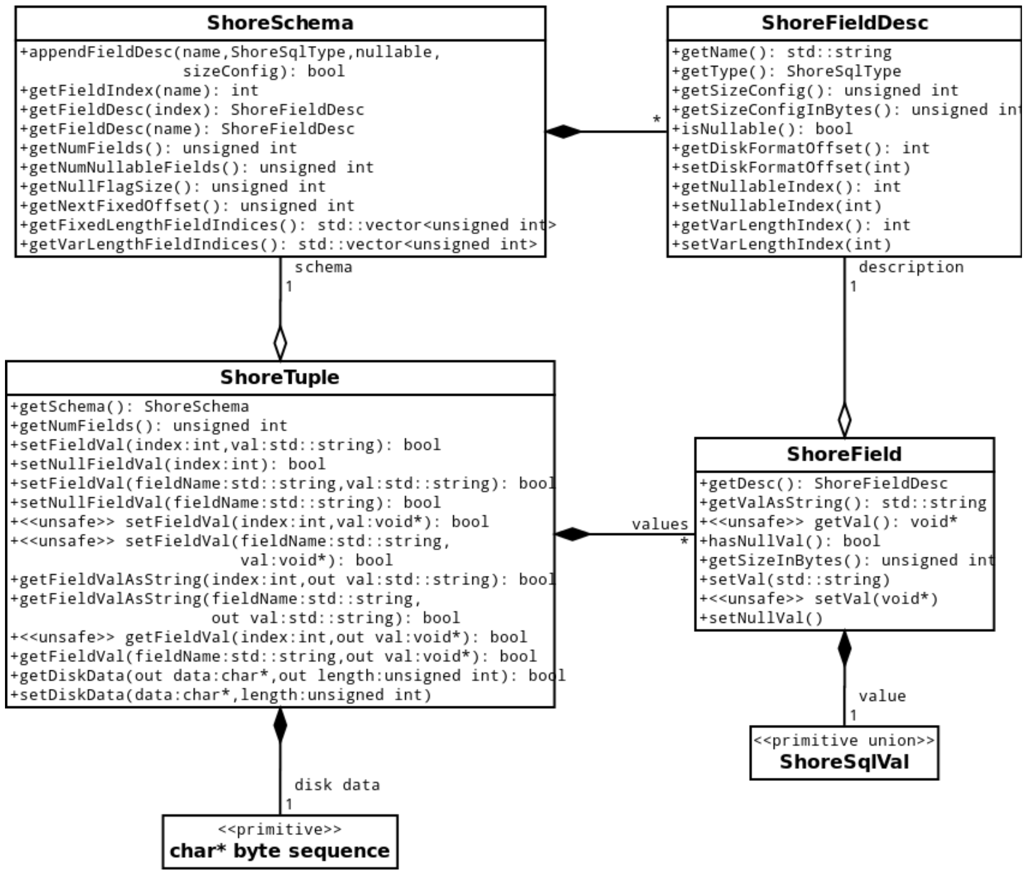

ShoreSchema
+appendFieldDesc(name,ShoreSqlType,nullable, sizeConfig): bool
+getFieldIndex(name): int
+getFieldDesc(index): ShoreFieldDesc
+getFieldDesc(name): ShoreFieldDesc
+getNumFields(): unsigned int
+getNumNullableFields(): unsigned int
+getNullFlagSize(): unsigned int
+getNextFixedOffset(): unsigned int
+getFixedLengthFieldIndices(): std::vector<unsigned int>
+getVarLengthFieldIndices(): std::vector<unsigned int>
ShoreFieldDesc
+getName(): std::string
+getType(): ShoreSqlType
+getSizeConfig(): unsigned int
+getSizeConfigInBytes(): unsigned int
+isNullable(): bool
+getDiskFormatOffset(): int
+setDiskFormatOffset(int)
+getNullableIndex(): int
+setNullableIndex(int)
+getVarLengthIndex(): int
+setVarLengthIndex(int)
*
schema
1
description
1
ShoreTuple
+getSchema(): ShoreSchema
+getNumFields(): unsigned int
+setFieldVal(index:int,val:std::string): bool
+setNullFieldVal(index:int): bool
+setFieldVal(fieldName:std::string,val:std::string): bool
+setNullFieldVal(fieldName:std::string): bool
+<<unsafe>> setFieldVal(index:int,val:void*): bool
+<<unsafe>> setFieldVal(fieldName:std::string, val:void*): bool
+getFieldValAsString(index:int,out val:std::string): bool
+getFieldValAsString(fieldName:std::string, out val:std::string): bool
+<<unsafe>> getFieldVal(index:int,out val:void*): bool
+getFieldVal(fieldName:std::string,out val:void*): bool
+getDiskData(out data:char*,out length:unsigned int): bool
+setDiskData(data:char*,length:unsigned int)
values
*
ShoreField
+getDesc(): ShoreFieldDesc
+getValAsString(): std::string
+<<unsafe>> getVal(): void*
+hasNullVal(): bool
+getSizeInBytes(): unsigned int
+setVal(std::string)
+<<unsafe>> setVal(void*)
+setNullVal()
value
1
<<primitive union>>
ShoreSqlVal
disk data
1
<<primitive>>
char* byte sequence

Figure 8: UML class diagram of relational data structures

First of all, a `ShoreFieldDesc` represents a field description and a `ShoreSchema` is an ordered collection of `ShoreFieldDescs`. The relationship between `ShoreSchema` and `ShoreFieldDesc` is naturally a strict containment or composition.

While `ShoreFieldDesc` is by and large meant for data encapsulation, `ShoreTuple` and `ShoreSchema` include functionalities. In particular, conversions from logical tuples to binary Shore-MT record format and vice versa (Section 4.3.4) are conducted by `ShoreTuple`. Certain offsets into the binary block are pre-computed by `ShoreSchema`.

As can be seen from Figure 8, `ShoreSqlVal` (introduced in Section 4.3.3) is an integral part of a `ShoreField` as the former cannot be used without SQL type information.

Also, it is worth highlighting that a `ShoreTuple` has a very **dynamic structure**. It strictly contains two alternative representations of a relational tuple: (1) disk data (formatted for physical record – see Section 4.3.4) and (2) an ordered list of logical `ShoreFields`. From an external perspective, it contains both representations at the same time and they are in sync. In reality, only relevant `ShoreFields` or disk data is materialised.

Lastly, certain operations are considered **type-unsafe** as they employ type-erased `void*` pointers to pass values of different system types. These are added for efficiency, and some have been used for internal operations. In future, they should remain solely for internal use unless the developer is absolutely sure.

The reader is highly recommended to refer to the comprehensive documentation in the source codes for further technical explanations.

4.3.6. Core object design & top-level API

Core object design of the Relational Data Manager (see Figure 9) refers to higher-level components which provide Shore-MT-related functionalities. They still represent relational concepts while using data structures introduced in Section 4.3.5 as a basis.

As depicted in Figure 9, two entities directly concern relational concepts and the functional design (Section 4.3.1), namely `ShoreTable` and `ShoreDatabase`. They act as the middlemen between the front-end and Shore-MT by abstracting away low-level storage structure details. This abstraction realises the relational-to-storage mappings discussed in Section 4.3.2 via Executor Service tasks described in Section 4.2.

Conceptually, every `ShoreTable` contains a collection of `ShoreTuples`; however, this is not a physical containment as `ShoreTuples` are only materialised on demand. On the other hand, `ShoreSchema` exhibits strict composition by `ShoreTable`.

An `Iterator` can be applied to any `ShoreTable` to sequentially scan its underlying storage file and materialise `ShoreTuples` along the way. A convenient method is also provided by `ShoreTable` to effectively retrieve all tuples in one go.

Last but not least, there is a `ShoreFrontend` component which is the **top-level API** of Shore-MT front-end. Users only need to instantiate and interact with `ShoreFrontend`

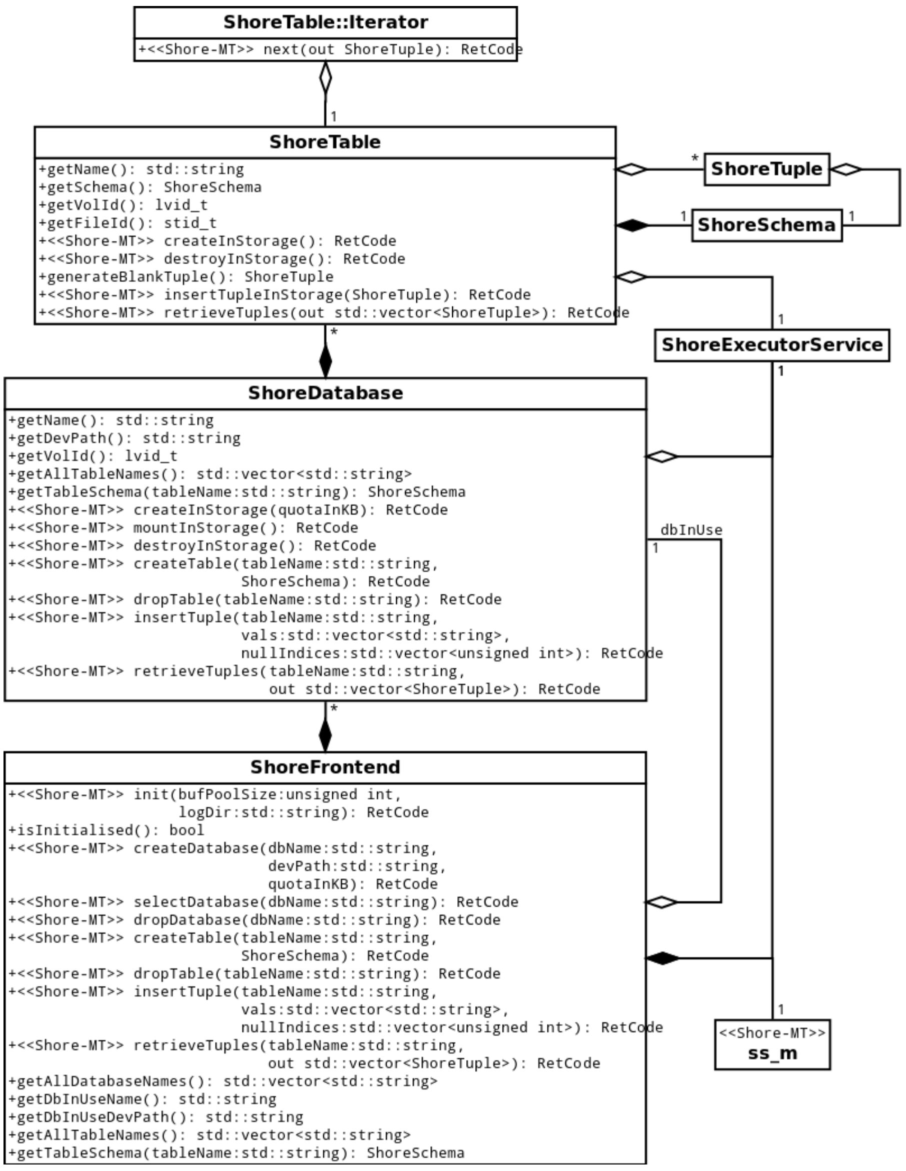

ShoreTable::Iterator
+<<Shore-MT>> next(out ShoreTuple): RetCode
1
ShoreTable
+getName(): std::string
+getSchema(): ShoreSchema
+getVolId(): lvid_t
+getFileId(): stid_t
+<<Shore-MT>> createInStorage(): RetCode
+<<Shore-MT>> destroyInStorage(): RetCode
+generateBlankTuple(): ShoreTuple
+<<Shore-MT>> insertTupleInStorage(ShoreTuple): RetCode
+<<Shore-MT>> retrieveTuples(out std::vector<ShoreTuple>): RetCode
*
ShoreTuple
1
ShoreSchema
1
1
ShoreExecutorService
1
*
ShoreDatabase
+getName(): std::string
+getDevPath(): std::string
+getVolId(): lvid_t
+getAllTableNames(): std::vector<std::string>
+getTableSchema(tableName:std::string): ShoreSchema
+<<Shore-MT>> createInStorage(quotaInKB): RetCode
+<<Shore-MT>> mountInStorage(): RetCode
+<<Shore-MT>> destroyInStorage(): RetCode
+<<Shore-MT>> createTable(tableName:std::string, ShoreSchema): RetCode
+<<Shore-MT>> dropTable(tableName:std::string): RetCode
+<<Shore-MT>> insertTuple(tableName:std::string, vals:std::vector<std::string>, nullIndices:std::vector<unsigned int>): RetCode
+<<Shore-MT>> retrieveTuples(tableName:std::string, out std::vector<ShoreTuple>): RetCode
dbInUse
1
*
ShoreFrontend
+<<Shore-MT>> init(bufPoolSize:unsigned int, logDir:std::string): RetCode
+isInitialised(): bool
+<<Shore-MT>> createDatabase(dbName:std::string, devPath:std::string, quotaInKB): RetCode
+<<Shore-MT>> selectDatabase(dbName:std::string): RetCode
+<<Shore-MT>> dropDatabase(dbName:std::string): RetCode
+<<Shore-MT>> createTable(tableName:std::string, ShoreSchema): RetCode
+<<Shore-MT>> dropTable(tableName:std::string): RetCode
+<<Shore-MT>> insertTuple(tableName:std::string, vals:std::vector<std::string>, nullIndices:std::vector<unsigned int>): RetCode
+<<Shore-MT>> retrieveTuples(tableName:std::string, out std::vector<ShoreTuple>): RetCode
+getAllDatabaseNames(): std::vector<std::string>
+getDbInUseName(): std::string
+getDbInUseDevPath(): std::string
+getAllTableNames(): std::vector<std::string>
+getTableSchema(tableName:std::string): ShoreSchema
1
<<Shore-MT>>
ss_m

Figure 9: Core object design of the Relational Data Manager

which encapsulates all functionalities described in Section 4.3.1. It internally owns a `ShoreExecutorService`, shares it with `ShoreDatabases` and `ShoreTables` and manages the underlying storage manager's lifetime.

In Figure 9, methods incurring interactions with Shore-MT are marked with "`Shore-MT`". They return a `RetCode` which has front-end's status code packed with the original Shore-MT status codes, if applicable. Once again, the reader is strongly advised to consult the comprehensive documentation in the source codes for in-depth technical descriptions.

4.4. Relational Metadata Manager

Relational Metadata Manager is an extension of the Relational Data Manager (Section 4.3). It augments the Relational Data Manager with metadata persistence. Relational metadata denotes meta-information of user's relational data such as directory of known databases, catalogues of tables and relational schemas.

4.4.1. Persisting relational metadata

The Relational Metadata Manager is designed as a module on top of the Relational Data Manager, reflectively employing the latter to persist metadata. In other words, metadata themselves are formulated as relational data and persisted onto Shore-MT storage using the facilities already in place. Figure 10 depicts a complete view of the storage layout with metadata persistence (cf. Figures 1 and 6).

In principle, there are two types of metadata: (1) catalogues of relational entities and (2) relational schemas. **Catalogues of relational entities** help locate databases and tables while **relational schemas** assist the interpretation of tuples retrieved from Shore-MT storage (which otherwise appear as meaningless blocks of binary data).

Catalogues of relational entities are persisted as follows:

- *Catalogue of databases*: A device is designated as **master device** (*not* a database as defined by the mappings of Section 4.3.2). There are no stores on this device except the volume root index which maps database names to device paths.
- *Catalogue of tables*: For each database device, the volume **root index** stores locations of constituent tables as mappings from table names to Shore-MT file IDs.

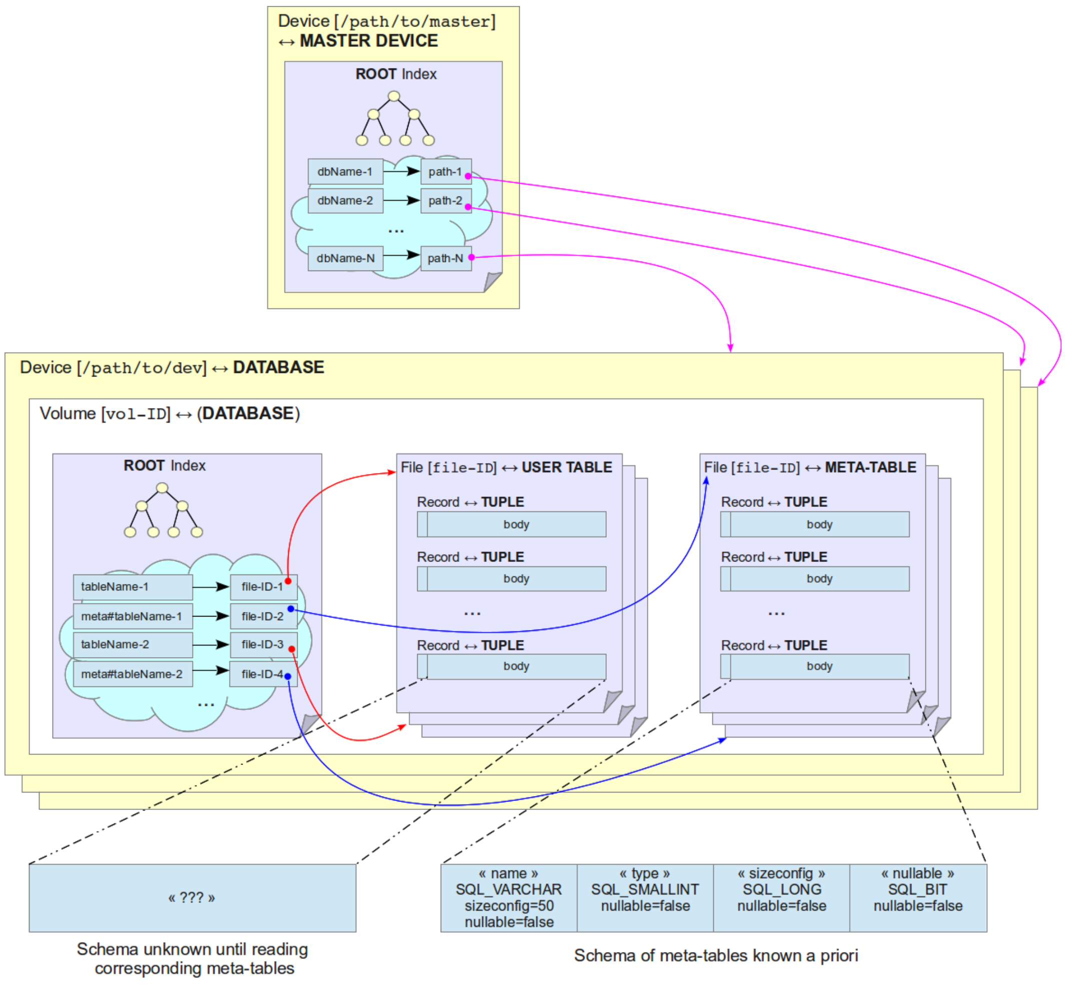
Relational schemas are persisted as follows:

- Every user table is associated with a **meta-table** whose schema ('meta-schema') is known a priori. Meta-tables store user schemas in the form of relational data.
- The **meta-schema** defines four fields for meta-table tuples: `name` (`VARCHAR`), `type` (`SMALLINT`), `sizeconfig` (`LONG`) and `nullable` (`BIT`). Each meta-table tuple represents one field description in the corresponding user table's schema.
- **Meta-table names** are corresponding user table names prefixed with `"meta#"` [9].

4.4.2. Object design & finalised top-level API

Only an extra sub-class `ShoreMetaTable` has to be defined while metadata persistence can be added to existing components like `ShoreDatabase` and `ShoreFrontend`. Figure 11 gives the final design while not replicating details already shown in Figure 9.

[9] As a result, no user table names are allowed to start with `"meta#"`.



Device [/path/to/master] ↔ MASTER DEVICE
ROOT Index
dbName-1
path-1
dbName-2
path-2
...
dbName-N
path-N
Device [/path/to/dev] ↔ DATABASE
Volume [vol-ID] ↔ (DATABASE)
ROOT Index
tableName-1
file-ID-1
meta#tableName-1
file-ID-2
tableName-2
file-ID-3
meta#tableName-2
file-ID-4
...
File [file-ID] ↔ USER TABLE
Record ↔ TUPLE
body
File [file-ID] ↔ META-TABLE
Record ↔ TUPLE
body
« ??? »
Schema unknown until reading corresponding meta-tables
« name » SQL_VARCHAR sizeconfig=50 nullable=false
« type » SQL_SMALLINT nullable=false
« sizeconfig » SQL_LONG nullable=false
« nullable » SQL_BIT nullable=false
Schema of meta-tables known a priori

Figure 10: Full view of Shore-MT front-end's storage layout

4.5. Demo application

An executable console application is provided to demonstrate and dynamically test functionalities of Shore-MT front-end. Instructions on how to run the application can be found in Appendix A.1. As indicated in the project scope (Section 1.5), no formal parsers of any standardised languages are made; instead, an ad hoc syntax is introduced for user inputs at the application's command prompt.

5. SWIG-ging into the scripting world

One project objective (Section 1.4) is to make Shore-MT front-end available in the Python environment [11] to facilitate prototyping from a Python console or scripting module.

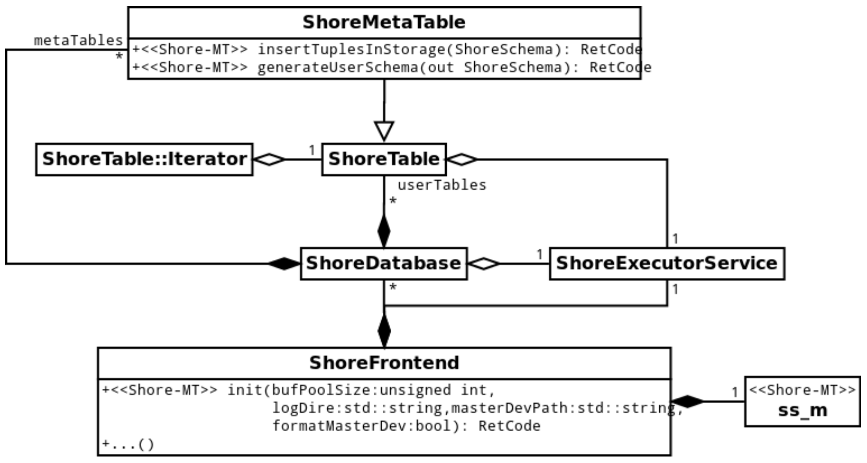

ShoreMetaTable
+<<Shore-MT>> insertTuplesInStorage(ShoreSchema): RetCode
+<<Shore-MT>> generateUserSchema(out ShoreSchema): RetCode
metaTables
*
ShoreTable::Iterator
1
ShoreTable
userTables
*
ShoreDatabase
1
ShoreExecutorService
1
*
1
ShoreFrontend
+<<Shore-MT>> init(bufPoolSize:unsigned int, logDire:std::string,masterDevPath:std::string, formatMasterDev:bool): RetCode
+...()
1
<<Shore-MT>>
ss_m

Figure 11: Core object design of the Relational Metadata Manager

5.1. Interfacing process

The process has been achieved with the SWIG interfacing library [12], version `1.3.29`. Figure 12 shows the procedure of wrapping Shore-MT for Python.

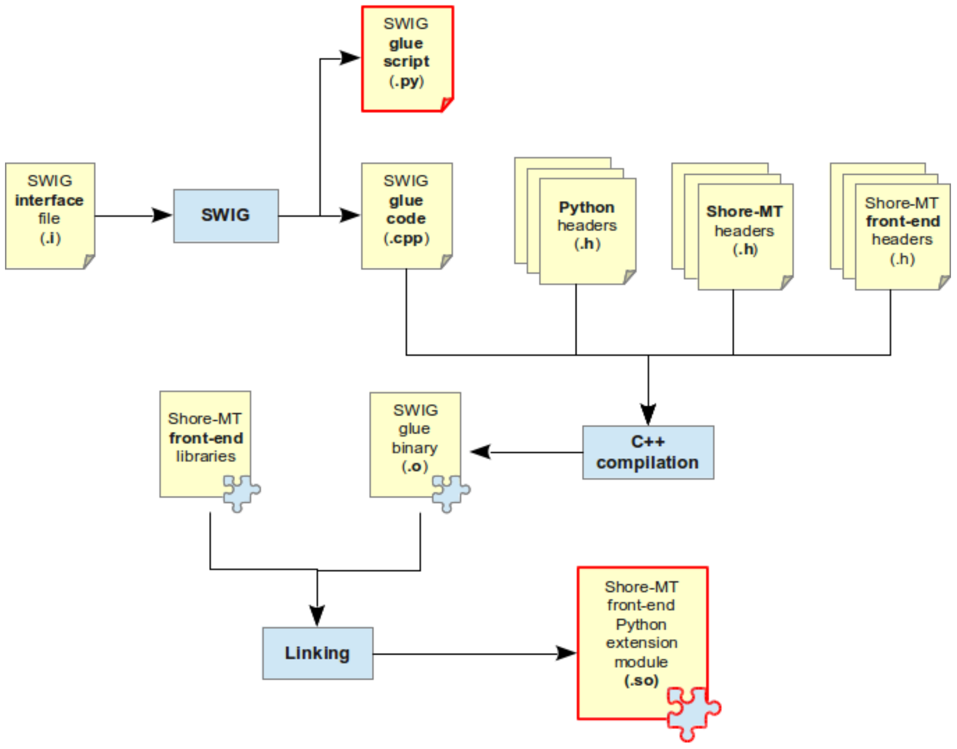


Figure 12: SWIG process to produce Shore-MT front-end Python extension

As presented in Figure 12, a **SWIG interface file** (`.i`) needs to be crafted for the wrapping process. Although SWIG interface files resemble C++ header files, they are *not*. Rather, they provide hints to SWIG for the generation of glue codes which must then be compiled with a standard compiler. The output of the whole process is a Python glue script (`.py`) and a native library called Python extension module (`.so`).

In this SWIG interface file, elements (classes, methods, attributes, etc.) to be exposed to the scripting environment have to be identified. In the case of Shore-MT front-end:

- Top-level API (where all functionalities are centralised): `ShoreFrontend`;
- Data structures required by top-level API (as parameter or return types): `RetCode`, `ShoreSqlType`, `ShoreFieldDesc`, `ShoreSchema` and `ShoreTuple`.

There have been no major hiccough in the process thanks to the fact that Shore-MT front-end design has been contemplated with scripting interfacing in mind. Some special SWIG directives have to be included so that C++ Standard Libraries [28] can be wrapped properly and certain Python particularities can function.

5.2. Usage in Python

All features of Shore-MT front-end (see Sections 1.4 and 1.5) are available as a Python module loadable via the `import` syntax. Furthermore, these features appear as natural Python API and the user needs not worry about low-level C++ aspects.

5.3. Python demo application

A Python demo module has also been developed to mimic the C++ one described in Section 4.5. As a matter of fact, the Python demo does not require command-prompt features and input parsing. Instead, the Python console is employed and for complex arguments, the user is supposed to construct them. Just as its C++ counterpart, the application has been used for dynamic testing from Python. Practical instructions regarding the Python extension module and demo can be found in Appendix A.2.

6. Conclusions

In sum, all objectives identified at the outset of the project (Section 1.4) have been satisfied with quality deliverables, in terms of design, functionality and transferability.

Apart from the Shore-MT front-end, a longed-for tool by DIAS researchers, the project provides a well-documented design, readable source codes and also a value-added reusable utility. However, there assuredly remains room for improvement to this preliminary piece of work. The following future work could be suggested:

- Complete separation and enhancement of the Executor Service utility module.
- Addition of index supports for user data.
- Implementation of SQL operators for prototyping of transaction processing.
- Generalisation of the SWIG interface for target languages other than Python.
- Incorporation of an SQL parser or the like.
- Facilitation of multi-threaded transactions at user level.

Acknowledgements

The student would like to thank Prof Anastasia Ailamaki for this interesting and challenging project opportunity with DIAS Laboratory at EPFL.

The student's gratitude also goes to DIAS research staff members, especially Pınar Tözün and Danica Porobic for the invaluable guidance and continuous support, and Miguel Branco and Radu Ioan Stoica for the stimulating inputs.

References

[1] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09. New York, NY, USA: ACM, 2009, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/1516360.1516365

[2] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling, "Shoring up persistent applications," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/191839.191915

[3] "SHORE project." [Online]. Available: http://research.cs.wisc.edu/shore/

[4] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A native XML database," *The VLDB Journal*, vol. 11, no. 4, pp. 274–291, Dec. 2002. [Online]. Available: http://dx.doi.org/10.1007/s00778-002-0081-x

[5] C. The Paradise Team, "Paradise: a database system for GIS applications," in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '95. New York, NY, USA: ACM, 1995, pp. 485–. [Online]. Available: http://doi.acm.org/10.1145/223784.223898

[6] L. Fegaras, C. Srinivasan, A. Rajendran, and D. Maier, "λ-DB: an ODMG-based object-oriented DBMS," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 583–. [Online]. Available: http://doi.acm.org/10.1145/342009.335494

[7] "Data-Intensive Applications and Systems (DIAS) Laboratory, EPFL." [Online]. Available: http://dias.epfl.ch

[8] "Shore-Kits." [Online]. Available: https://sites.google.com/site/shoremt/source-code/experimental

[9] "Transaction Processing Performance Council (TPC)." [Online]. Available: http://www.tpc.org

[10] R. Ramakrishnan and J. Gehrke, "The relational model," in *Database Management Systems*, ser. McGraw-Hill international editions: Computer science series. McGraw-Hill Companies, Inc., 2002, ch. 3. [Online]. Available: http://books.google.ch/books?id=JSVhe-WLGZ0C

[11] "Python programming language." [Online]. Available: http://www.python.org

[12] "Simplified Wrapper and Interface Generator (SWIG)." [Online]. Available: http://www.swig.org

[13] C. The Shore Project Group, "Storage manager architecture," University of Wisconsin-Madison, Tech. Rep., Jun. 1999.

[14] ——, "The SHORE storage manager programming interface," University of Wisconsin-Madison, Tech. Rep., Aug. 1999.

[15] "SHORE storage manager: The multi-threaded version." [Online]. Available: http://research.cs.wisc.edu/shore-mt/onlinedoc/html/main.html

[16] "Structured Query Language (SQL)." [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498

[17] "Shore-MT project." [Online]. Available: https://sites.google.com/site/shoremt/home

[18] R. Ramakrishnan and J. Gehrke, "Tree-structured indexing," in *Database Management Systems*, ser. McGraw-Hill international editions: Computer science series. McGraw-Hill Companies, Inc., 2002, ch. 10. [Online]. Available: http://books.google.ch/books?id=JSVhe-WLGZ0C

[19] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/93597.98741

[20] B. Bruegge and A. Dutoit, *Object-oriented software engineering: using UML, patterns and Java*. Prentice Hall, 2003. [Online]. Available: http://books.google.ch/books?id=26ZQAAAAMAAJ

[21] "Unified Modelling Language (UML) 2.0." [Online]. Available: http://www.omg.org/spec/UML/2.0

[22] "GNU Autoconf." [Online]. Available: http://www.gnu.org/software/autoconf

[23] "GNU Automake." [Online]. Available: http://www.gnu.org/software/automake

[24] "GNU Libtool." [Online]. Available: http://www.gnu.org/software/libtool

[25] "GNU Make." [Online]. Available: http://www.gnu.org/software/make

[26] "GNU Compiler Collection (GCC)." [Online]. Available: http://gcc.gnu.org

[27] "Mercurial source control management system." [Online]. Available: http://mercurial.selenic.com

[28] B. Stroustrup, "Part III: The Standard Library," in *The C++ programming language*. Boston: Addison-Wesley, 2000, ch. 16–22.

[29] "Doxygen." [Online]. Available: http://doxygen.org

[30] "Java SE 6 – ExecutorService." [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html

[31] "Boost C++ libraries." [Online]. Available: http://www.boost.org

[32] B. Stroustrup, "Types and declarations," in *The C++ programming language*. Boston: Addison-Wesley, 2000, ch. 4.

A. Appendix: Set-up & build

A.1. Main C++ front-end

As an HG-tracked project, Shore-MT front-end can be checked out to any directory from its repository URL by executing HG `clone` command, which by default updates the project to its latest revision. More help on HG commands can be found at [27]:

```
$ hg clone <repository-URL>
```

It should be noted that all C++ source files (header `.h` and implementation `.cpp` files) are located in a sub-directory called `src`. The `autogen.sh` shell script found in the project can then be run to generate configuration and build scripts, according to GNU Autotools **build system** [22–24]:

```
$ ./autogen.sh
```

Next, the project can be configured with current Shore-MT installation directory[10] and desired installation directory for Shore-MT front-end, by running the generated `configure` shell script:

```
$ ./configure SHORE_HOME=/path/to/shore-mt/installation
              --prefix=/path/to/front-end/installation
```

Now standard GNU Make [25] commands can be used to invoke GCC [26]. The following clean, build and install Shore-MT front-end, respectively:

```
$ make clean
$ make
$ make install
```

The installation of Shore-MT front-end should be similar to the following. The executable demo application mentioned in Section 4.5 is `bin/shore-frontend`:

[10]This refers to the location of `make install` artefacts when building Shore-MT.

```
|-- bin
|   |-- shore-frontend
|-- include
|   |-- <header .h files>
|-- lib
    |-- libshore-frontend.a
    |-- libshore-frontend.la
    |-- libshore-frontend.so -> libshore-frontend.so.0.0.0
    |-- libshore-frontend.so.0 -> libshore-frontend.so.0.0.0
    |-- libshore-frontend.so.0.0.0
```

Once the demo application is started, the user is greeted with a command prompt. Listing of commands and their usage can be shown by running `help`:

```
$ cd /path/to/front-end/installation/bin
$ ./shore-frontend
cpu_info sees 4 sockets and 16 cores

*** Welcome to Shore-MT frontend console! ***
(Type 'help' to list available commands)

$ help
Shore-MT frontend command list:

init <bufPoolSize> <logDir> <masterDevPath> <formatMasterDev?>
- Initialise Shore-MT frontend

init? - Check if Shore-MT frontend has been initialised
show-dbs - Show all available databases

create-db <dbName> <devPath> <quotaInKb> - Create a new database
select-db <dbName> - Select an existing database for use
db-in-use? - Show info of selected database
drop-db <dbName> - Drop an existing database

create-table <tableName> <numFields>
[<fieldName> <type> <nullable?> <sizeConfig>...] - Create a new table

drop-table <tableName> - Drop an existing table
show-tables - Show all tables in selected database
show-schema <tableName> - Show schema of table

insert-tuple <tableName> <numFields>
[<fieldVal_1> <fieldVal_2>...] - Insert a tuple

retrieve-tuples <tableName> - Retrieve all tuples

help - Show this list of commands
quit|exit - Quit Shore-MT frontend console
```

A.2. Python interfacing module

The Python interfacing project of Shore-MT front-end can be checked out from its HG repository URL in a similar manner. In this project, the SWIG interface file (as described in Section 5.1) is `shore_frontend_swig.i` located at the root directory. Five convenient shell scripts are provided to build the interfacing module (cf. Figure 12):

- `clean.sh`: Clean all build artefacts.
- `swig.sh`: Generate SWIG C++ and Python glue codes from the SWIG interface file.
- `compile.sh`: Compile the C++ glue code. Required environment variables:
  - `SHORE_FRONTEND_HOME`: Path to Shore-MT front-end installation[11].
  - `SHORE_HOME`: Path to Shore-MT installation[12].
  - `PYTHON_INCLUDE`: Path to Python header files.
- `link.sh`: Link the resultant binary object with Shore-MT front-end libraries to produce the final Python extension module. Required environment variable:
  - `SHORE_FRONTEND_HOME`: Path to Shore-MT front-end installation[13].
- `all.sh`: Sequentially execute `clean.sh`, `swig.sh`, `compile.sh` and `link.sh`.

The final run-time artefacts are Python glue script `shore_frontend.py` and Python extension module `_shore_frontend.so`. Provided that all necessary libraries are available on the system library path, one way to load Shore-MT front-end in Python is as follows:

```
from shore_frontend import *
```

Similarly, the demo (provided as `shore_frontend_prog.py`) can be loaded with the following statement in Python:

```
from shore_frontend_prog import *
```

B. Appendix: File and class listings

B.1. Top-level components

C++ class/executable	Header file	Implementation file
Demo application		`shore-frontend-prog.cpp`
`ShoreFrontend`	`shore-frontend.h`	`shore-frontend.cpp`
`RetCode`	`shore-frontend-rc.h`	`shore-frontend-rc.cpp`

[11]Directory of `make install` artefacts, having header files in a sub-directory called `include`.
[12]Directory of `make install` artefacts, having header files in a sub-directory called `include`.
[13]Directory of `make install` artefacts, having library binaries in a sub-directory called `lib`.

B.2. Executor Service module

C++ class	Header file	Implementation file
`ShoreExecutorService`	`shore-executor-service.h`	`shore-executor-service.cpp`
`ShoreWorkerThread`	`shore-worker-thread.h`	`shore-worker-thread.cpp`
`ShoreTask`	`shore-task.h`	`shore-task.cpp`
`ShoreTaskCompletionListener`	`shore-task-completion-listener.h`	

B.3. Front-end specific tasks

C++ class	Header file	Implementation file
`BaseTask`	`shore-frontend-base-task.h`	`shore-frontend-base-task.cpp`

`SmTask`
`CreateSmTask`
`DestroySmTask`
	`shore-sm-tasks.h`	`shore-sm-tasks.cpp`

`BeginTransactionTask` `EndTransactionTask`	`shore-transaction-tasks.h`	`shore-transaction-tasks.cpp`

`DevTask`
`MountDevTask`
`InitDevTask`
`DestroyDevTask`
	`shore-dev-tasks.h`	`shore-dev-tasks.cpp`

`FileTask`
`CreateFileTask`
`DestroyFileTask`
`CreateRecordTask`
	`shore-file-tasks.h`	`shore-file-tasks.cpp`

`RootIndexTask`
`DestroyRootIndexEntriesTask`
`CreateRootIndexEntryTask`
`RetrieveAllRootIndexTask`
	`shore-root-index-tasks.h`	`shore-root-index-tasks.cpp`

B.4. Data Manager modules

C++ class/type	Header file	Implementation file
`ShoreDatabase`	`shore-database.h`	`shore-database.cpp`
`ShoreTable`	`shore-table.h`	`shore-table.cpp`
`ShoreMetaTable`	`shore-metatable.h`	`shore-metatable.cpp`
`ShoreSchema`	`shore-schema.h`	`shore-schema.cpp`
`ShoreFieldDesc`	`shore-field-desc.h`	`shore-field-desc.cpp`
`ShoreTuple`	`shore-tuple.h`	`shore-tuple.cpp`
`ShoreField`	`shore-field.h`	`shore-field.cpp`
`ShoreSqlVal`	`shore-sql-val.h`	`shore-sql-val.cpp`
`ShoreSqlType`	`shore-sql-type.h`	`shore-sql-type.cpp`