A PHYSICAL STORE FOR A RELATIONAL DATABASE IN THE STL STYLE

TALAL AL-KHOURY

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

JUNE 2003

Your file Votre référence
ISBN: 0-612-83894-3
Our file Notre référence
ISBN: 0-612-83894-3

Abstract

A Physical Store for a Relational Database in the STL Style

Talal Al-khoury

We introduce an object-oriented design of a physical store system for a relational database. The design and implementation of the physical store are in the style of C++ standard template library. They stress separation of memory allocation, iterators, container types, and the types of data in the store. The physical store system is part of the Know-It-All database project, under way at Concordia University. The contents of the physical store system are described using SQL data definition language. The main tasks of the physical store system are the bulk- loading of data, the support for the bulk creation of indexes, and data retrieval. The design hides the use of the secondary store. The problems with the existing physical stores are a lack of type checking and type safety. Our design and implementation draw upon advanced uses of C++ templates, as typified by the standard template library, type lists, the tuple library from boost.org, and dynamic loading. Furthermore we reuse design patterns, the mySQL parser, and the POST++ persistent object store.

Acknowledgements

I would like to express my sincere respect and gratitude to my thesis supervisor Dr. Gregory Butler for his guidance, encouragement and support through the course of my research work. I would like to thank Dr Gregory Butler also for the financial support.

I would like to thank my friends for their discussion, suggestions, comments and help on my thesis.

I would like to thank my new country Canada for financial support.

I would like to thank Concordia University for giving me this chance.

Contents

List of Figures

List of Tables

1. Introduction

In this chapter, we introduce the problem that our work deals with, the issues of reusable design and the style of the C++ Standard Template Library, the contribution of the thesis and the organization of the thesis.

In our work, we introduce an object-oriented design of a physical store system for a relational database. The physical store is a component of a prototype relational database being constructed as an instance of the Know-It-All framework for database technology.

At this stage, we are not designing a subframework for physical storage, just an example of a physical store for relational data. Nevertheless, the design should conform to the existing architecture and style of the KIA framework. The architecture is in terms of layers, the style is heavily influence by the C++ Standard Template Library, and an indexing subframework in this style was the work of a previous masters thesis [AG 01]. The indexing subframework was based on the GIST system from Berkeley [HNP95].

The basic requirements of the physical store are to fit into the context as designed by the indexing subframework and to support the bulk loading of data, the bulk creation of indexes, and the retrieval of data. Incremental data loading was an optional extension. Transaction support was not a requirement, though it was hoped to include locks in a natural way.

Gaffar's work separated the concepts of Data and DataRef [AG 01]. An index shared DataRefs, which were handles or proxies to the actual data. The physical store could dereference a DataRef object in order to retrieve the corresponding Data object. Bulk loading of data required the ability to read a schema description given in the data definition language of SQL, to load the data, and to provide the indexing subframework

with the corresponding information about keys and DataRef object-pairs so that the indexes can be created in bulk.

The problems of the existing physical stores that are implemented in the C programming language are mainly the lack of type checking and type safety. Our main issues were problem of design, i.e. using the most modern abilities of object oriented design and C++ templates for reusability and extendibility. This required reading many books and much source code. To develop high quality software, we use number of technique and tools such as the typlists to generate tuples classes dynamically, the parser of mySQL to compile the data definition language of SQL of the user's commands, the BOOST library to use its implementation of tuple class and the POST++ library to use its implementation of persistent allocator.

Dr. Gregory Butler is interested in the database domain as a case study to validate a framework methodology. A number of thesis' researches have been achieved under the supervision of Dr. Gregory Butler in the field of framework methodologies and development for scientific applications. The results of these researches is the ongoing development of the Know-It-All framework for database management system that supports a variety of data models of data and knowledge, the integration of different paradigms and heterogeneous database [BCC02]. The project involves multiple subframeworks for physical storage, query optimization, index trees, diagrammatic query interface and others. These subframeworks and subsystems are integrated together in an adaptable DBMS context. It started by supporting the traditional relational database model, and it will expand to support other data models.

1.1. The Problem

The problem that we deal with, in our work, is the design and the implementation of the physical store system. The physical store should be compatible with previous subframeworks such as the index framework. An efficient physical store system is a fundamental requirement for good performance of database systems. Besides efficiency, we wish the physical store to provide type-safe storage and retrieval of data. The existing physical stores implemented in the C programming language lack type checking and type safety. Even though this thesis is not designing a subframework for physical data storage, we do care about flexibility and reusability of the design. The physical store should provide a friendly tool for the user to express his commands against the physical store. We believe the data definition language expressed by SQL is suitable for our project. Beside that, the physical store should be able to create a new database and process an existing database, create a new table in a database and process an existing table and to have an efficient algorithm to achieve a bulk load, and inserting data. Many applications need to load large amount of data from a source to a target system. One of the major components of the physical store system is the bulk load. The major difficulty in bulk loading is to optimize the loading process that may last hours or even days when data is huge. Another important issue is the ability to resume a load after a failure in order not to lose too much effort and time. Finally, a good loading system should organize data on disk so as to optimize the run time of the applications that will use it.

1.2. The Reusable Design and the STL Style

The experience of a good designer plays the major role in producing a high quality reusable design and code. However the reuse is difficult because of finding the proper abstraction for reuse. A rule of thumb of reuse is that you are not knowledgeable enough to build a reusable artifact until you have built three instances and you do not know your artifact is reusable until you have used it successfully three times. So we will not claim to achieve reusability, nor attempt it. However, we have studied more than three instances of physical stores. The pattern design helps in reuse by designing patterns such as proxy, adapter, visitor, strategy, Iterator etc, and if we face similar problems to these have been solved in these patterns, we can use the same solution, code and design proposed by these patterns. In our work we use smart pointers, iterators, factory and proxy as a method to achieve reuse.

Choosing the programming language to implement a very complex system is essential to achieve reusability and high quality performance. **The C++ language is the de facto standard in object-oriented design programming**. C++ has features like inheritance and polymorphism, but effective comparability and efficient execution favours the use of templates, delegation, and conformity to interfaces in preference to traditional generalization and polymorphism. The Standard Template Library (STL) is an example of the effective use of templates and separation of concerns. The STL library provides the ability to use, string types, different sorting algorithms, numeric classes, classes for internationalization and different data structures such as dynamic arrays, linked lists, and binary trees. We can build many complete sophisticated modern systems with the majority of its building blocks obtained from the STL library. A large number of building

blocks already exist with a complete implementation on hand. This dramatically reduces the time needed for the implementation for many large systems where a great percentage of the code is simply imported from the STL.

1.3. Contribution of the Thesis

This thesis introduces the design of the physical store, the prototype implementation of the design, and the complete implementation of a number of components of the physical store of a relational database. We implemented the system using STL. The design and implementation are original; through it we reuse some classes from the Boost tuple library [BOOST], POST++ persistent store library [POST] and Loki library [A02]. In our design, we consider reusability, flexibility and extensibility of the physical store implementation. For example, extending the system for multi-user and distributed databases.

The major concerns are:

- The definition of the specification of the physical store.
- The design of all components of the physical store.
- The implementation of a prototype of the physical store, which includes the implementation of the persistent store, which is compatible with the index subframework [AG 01].
- Using STL containers, algorithms and iterators to ensure efficient implementation, reusability, extensibility and understandability.
- Using the smart pointer technology to retrieve a tuple from the secondary store and to maintain the reference to the loaded tuple. The smart pointer is able to

control the locks on reading and writing from and into the file. So we can use the smart pointer to implement concurrency control.

- Using the Typelists technology [A02] to dynamically create the user tables classes and the keys classes.
- Using Factory method design pattern to control the creation of different kinds of buffers to customize the need of the client.
- Describing the design of the bulk load and considering the efficiency and flexibility issue when loading data from text file into a relational database.

We believe that the documentations of the physical store system are very important since the physical store is a part of a larger project. These documents include:

- The system overview that describes the context of the physical store system components.
- The physical store system design that describes the architectural design and the detailed design of the system.
- The detailed implementation of the main classes in the system and the concepts stand behind the implementation.
- The testing documentation for the components, modules and procedures that we implemented in the physical store system.

1.4. Organization of the Thesis

The thesis is divided into five chapters.

Chapter 2 provides a background of theories and concepts that we use in designing and implementing the physical store system such as the role of physical store in a traditional DBMS, the data definition languages that are used in the relational database, object

database and object-relational database. Then we introduce the characteristics of the STL style that are important for the design and implementation such as containers, allocators, algorithms and adapters. Then we introduce the tools and techniques that we use in implementing and designing physical store system such as the typelists, Boost tuple library, mySQL parser and POST++ persistent store. Then we introduce the major design patterns that are important to the physical store such as the factory and proxy patterns. The UML language that we use to document the physical store is also introduced.

Chapter 3 provides the specification of the physical store. This chapter starts by providing a description of the Know It All project and explaining how its components influence the physical store. Then it introduces the main functions of the physical store and the specification of the data definition language used in the physical store. The Bulk-loading Algorithm is introduced. After that the application program interfaces (API) of the physical store and the Physical Store Data Model are presented.

Chapter 4 introduces the architecture of the physical store and presents an object-oriented design for the physical store system. We discuss the structural aspects of this project, the overall structure, and the detailed design. We introduce the implementation of some classes of the system and the theory concepts that we use in our implementation. The scenario how the client interacts with the components of the system is also presented.

Chapter 5 provides the conclusion of our work, the future work and a user manual for some implemented procedures to test some classes in the system.

2. Background

In this chapter we provide a background of theories and concepts that we use in designing and implementing the physical store system such as the role of the physical store in a traditional DBMS, the data definition languages that are used in the relational databases, object database and object-relational databases. Then we introduce the characteristics of the STL style that are important for the design and implementation such as containers, allocators, algorithms and adapters. Then we introduce the tools and techniques that we use in implementing and designing physical store system such as the typelists, Boost tuple library, mySQL parser and POST++ persistent store. Then we introduce the major design patterns that are important to the physical store such as the factory and proxy patterns. The UML language that we use to document the physical store is also introduced.

2.1. Role of Physical Store in a Traditional DBMS

In this section, we introduce the role of the physical store in a traditional database management system. Figure 1 shows the structure (with some simplification) of a typical DBMS based on the relational data model [RG 00]. Figure 1 shows that the DBMS accepts SQL commands; produces query evaluation plans, executes these plans against the database, and returns the answers. The query is parsed and presented to a query optimizer to produce an efficient execution plan for evaluating the query. The files and access methods layer sites on top of the buffer manager, which brings pages in from disk to main memory as needed in response to read requests. The lowest layer of the DBMS

software deals with management of space on disk, where the data is stored. The DBMS supports concurrency and crash recovery that include the transaction manager, the lock manager, and the recovery manager.

The parts of relational database management system that we discuss, in this section, are the components of the physical store including the file system, the file manager, the buffer, the buffer manager, the catalogue and the lock manager. The basic functionality of the physical store and concurrency are discussed. To explain the basic functionality we introduce some examples from well-known systems such as mySQL, Shore and others.

2.1.1. File Structure

The main reason behind file structure design is that the secondary storage is very slow compared with the speed of the microprocessor and its random access memory (RAM). In order to increase the throughput of the operating system, we need to design a file structure that decreases the time access to the disk. The physical file exists in secondary storage; however, in order to use the physical file by a software system we have to represent this file in a logical way. This representation is the logical file. Our programs only see the logical file. In order to use the physical file, we need to implement some operations that deal with the creation, opening, closing of a file, reading from a file, and even finding a certain location in a file. We usually implement these operations in our system by making a system call (i.e. to the operating system) since the operating system does not allow our programs to run Input/ Output (I/O) operations directly for the safety of the system.

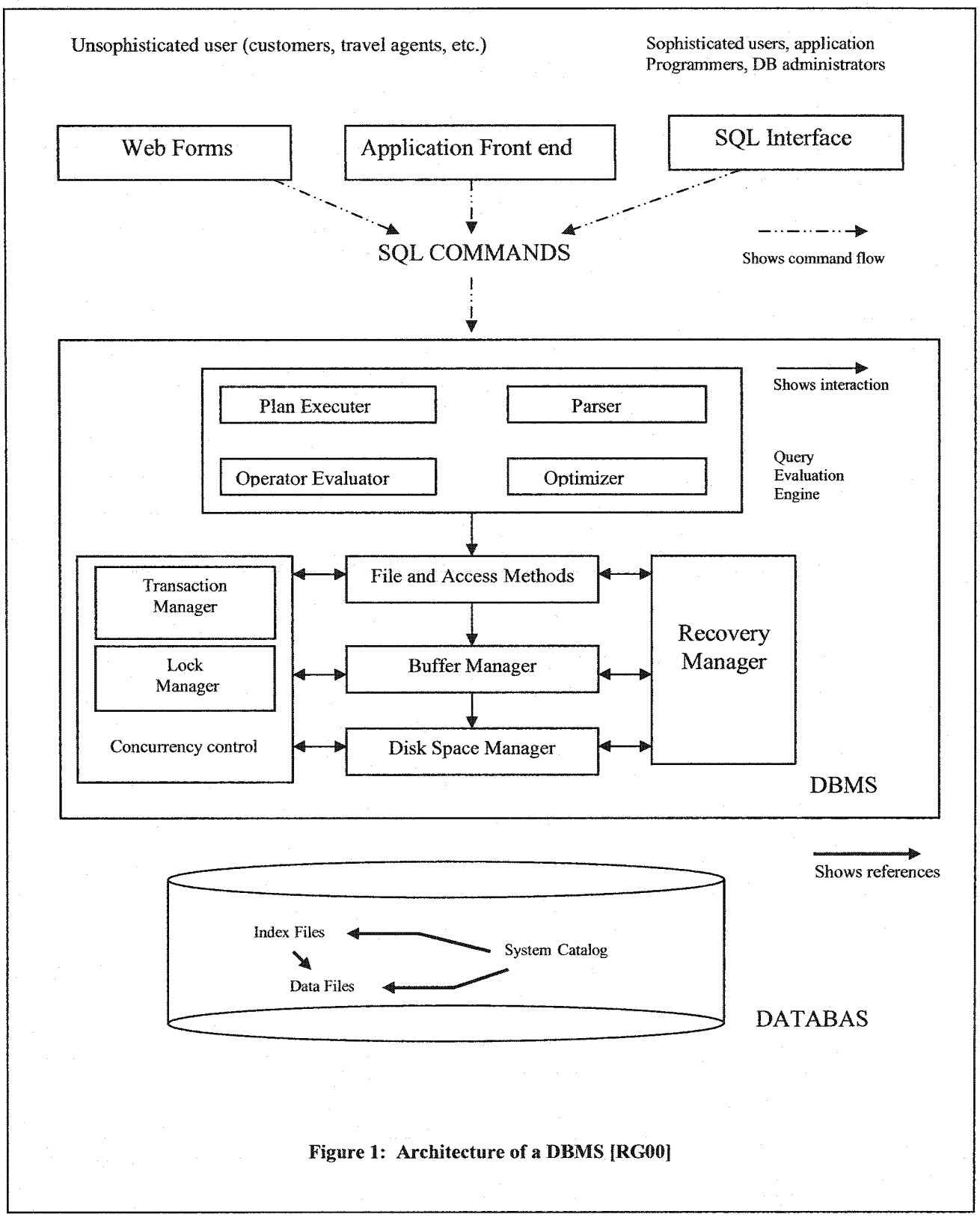


Figure 1: Architecture of a DBMS [RG00]

For example, Table 1 shows the file commands that most operating systems support.

Table 1: File Commands

Command name (parameters)	Meaning of the command
Create (file name)	creates a physical file in the secondary storage with the given physical file name. It returns the file descriptor of type integer.
Open (file name, flags [, pmode]);	makes the file with the given name ready for use. The flags parameter of type integer indicates how the file should be accessed whereas pmode of type integer specifies the protection mode for the file. It returns the file descriptor of type integer.
Close (descriptor)	breaks the links between the logical file and the corresponding physical file.
Read (source_file, Destination_addr, Size)	obtains input from a file. The parameters indicate the address in the file and the number of bytes to be brought consequently.
Write (Destination_addr, source_file, Size)	provides output to a file. The parameters indicate the address in the file and the number of bytes to be brought consequently.
Seek (source_file, offset)	sets the read/write pointer to a special position in a file.

2.1.2. The File Manager

The file manager is necessary, in computer systems, to achieve concurrency with the other parts of the system that can be run in the same time. Usually, the CPU is switched

between all the components of the system to improve the response time and the throughput of the system. The file manager receives, through calls to the operating system, our tasks to read to or write from a file. The file manager finds out whether the logical characteristics of the file are consistent with the task of our programs. The file manager uses some structures located on the disk, such as the file allocation table (FAT), to obtain the needed information about the physical location of the sectors related to the specific file, for example, the last sector in the file.

2.1.3. The Buffer

In order to optimize the number of I/O operations, the system does not send the data directly to the physical file. The data waits for some time in the buffer because the file manager wants to see if it is possible to accumulate more data in the buffer before making a decision to send the data to the disk. However this is not always possible. For example, if our programs give a task to close the file, the system should flush all output buffers holding data to be written to the file. In this case, the data does not wait any more in the buffer and it will be transferred to the disk to make sure that the data would not be lost.

2.1.4. The Buffer Manager

The buffer manager in the system tries to reduce the number of I/O operations. However, if we design, in our programs, our own buffers and buffer manager, we may increase the performance substantially. For example, we can improve the performance by using a multiple buffering strategy. In this strategy, when one of the buffers is used to perform an

I/O operation, the CPU can be used to fill another buffer with data and to prepare it for the next I/O operation. Another technique to improve performance is the Scatter / Gather I/O strategy which scatters the incoming data into a number of buffers on input and gathering the data from different buffers and organize them to be one output. Since the buffers area of the system and the data area of the programs are parts of the main memory, moving a big chunk of data between these two areas also improve the performance. This technique is called the Move mode strategy. In the locate mode strategy, we avoid the move mode either by transferring the data between the data area of the programs and the secondary storage directly or by providing a pointer to the location of the system buffer.

2.1.5. Transaction Management

The part of the system that controls the transactions is the transaction manager. The transaction manager provides strategies to implement a number of transactions concurrently and to produce the same results as if the transactions were processed separately one by one. The average number of transactions completed in a given time is called the system throughput. The transactions in the DBMS should have the following properties:

1. Consistency: The user is responsible for making his database consistent by submitting transactions that ensures that starting from a consistent database and after completing his transaction, the database moves to another consistent state.
2. Isolation: The transactions manager should ensure that the interleaved transactions are protected from each other. For example, let us to assume that the

scheduler of the transaction manager has two transactions T1 and T2; and T1 has the following actions: Read1 A, Write1 A, Read1 B, Write1 B; and T2 has the following actions: Read2 C, Write2 C. If the scheduler interleaves these actions, the result should be the same as if these transactions are implemented sequentially one by one, let say like the following order: Read1 A, Write1 A, Read1 B, Write1 B, Read2 C, Write2 C.

3. Atomicity: the transactions manager should ensure that the actions of a transaction should be implemented completely. Either all the actions of the transaction are implemented completely or the transaction is aborted. Otherwise, the database will not be in a consistent state and all the following transactions will cause more harm to the database. Sometimes, the transaction is not completed for some reasons such as it is aborted, the system crashes or it counters reading unexpected data value or it can not access some disk. In such case, the DBMS should eliminate all the actions that have been implemented against the database. A DBMS uses a log file to keep a trace of the transaction. To eliminate an incomplete transaction, DBMS reads the transaction trace from the log file and undoes the actions.
4. Durability: If the system crashed before moving the data to the disk and the user has been informed that his transaction is successful, it is the responsibility of the DBMS to complete the transaction. The DBMS also uses the log file to read the trace of the transaction and complete the implementation successfully. The part of the software that is responsible for ensuring the atomicity and the durability is called the recovery manager.

We usually use the acronym ACID to refer to the properties above.

2.1.6. Describing and Storing Data in a DBMS

A data model is a collection of high-level data description constructs that hide many low-level storage details. A semantic data model is a more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise.

In the relational model, the central data description construct is a relation, which can be thought as a set of records. A description of data in terms of a data model is called a schema. In the relational model, the schema for a relation specifies its name, the name of each field (or attribute or column), and the type of each field.

The data in a DBMS is described at three levels of abstraction, the conceptual, physical, and external. The conceptual schema describes the stored data in terms of the data model of the DBMS. The physical schema specifies storage details. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices. External schemas, which usually are also in terms of the data model of the DBMS, allow data access to be authorized and customized at the level of individual users or groups of users. Any given database has exactly one conceptual schema and one physical schema because it has one set of stored relations, but may have several external schemas, each tailored to particular group of users.

2.1.7. System Catalog in a Relational DBMS

We can store a relation using one of several alternative file structures, and we can create one or more indexes – each stored as a file – on every relation. Conversely, in a relational DBMS, every file contains either the tuples in a relation or the entries in an index. The collection of files corresponding to users' relations and indexes represents the data in the database.

A relational DBMS maintains information about every relation and index that it contains. The DBMS also contains information about the views. This information is stored in a collection of relations, maintained by the system, called the catalog relation; an example of a catalog relation is shown in Table 2. The catalog relation is also called the system catalog, catalog, data dictionary or metadata. The information in the system catalog is used extensively for query optimization.

Information Stored in the System Catalog

At a minimum we have system wide information, such as the size of the buffer pool and the page size, and the following information about individual relations, indexes, and views:

- For each relation:
    - Its relation name, the file name (or some identifier), and the file structure (e.g., heap file) of the file in which It is stored.
    - The attribute name and type of each of its attributes.
    - The index name of each index on the relation.
    - The integrity constraints on the relation.

- For each index:
    - The index name and the structure of the index.
    - The search key attributes.
- For each view
    - Its view name and definition.
- Cardinality: The number of tuples for each relation.
- Size: The number of pages for each relation.
- Index cardinality: Number of distinct key value for each relation.
- Index size: The number of pages for each index.
- Index height: The number of nonleaf levels for each tree index.
- Index range: The minimum percent key value and the maximum percent key value for each index.

In System R, a page file can contain pages that store records from more than one record file [RG 00]. If such a file organization is used, additional statistics must be maintained, such as the fraction of pages in a file that contain records from a given collection of records. The catalogs also contain information about users, such as accounting information and authorization information.

The catalog relation, describe all the relations in the database, including the catalog relations themselves. When information about a relation is needed, it is obtained from the system catalog. Of course, at the implementation level, whenever the DBMS needs to find the schema of a catalog relation, the code that retrieves this information must be handled specially. (Otherwise, this code would have to retrieve this information from the catalog relation without, presumably, knowing the schema of the catalog relations!)

The fact that the system catalog is also a collection of relations is very useful. For example, catalog relations can be queried just like any other relation, using the query language of the DBMS! Further, all the techniques available for implementing and managing relation apply directly to catalog relations.

Table 2: An Instance of the Catalog Records Relation

AttrName	RelName	Type	Position
AttrName	catalogRecords	string	1
RelName	catalogRecords	string	2
Type	catalogRecords	string	3
Position	catalogRecords	int	4
Sid	Students	string	1
Name	Students	string	2
Login	Students	string	3
Age	Students	int	4
Gpa	Students	double	5
Fid	Faculty	string	1
Fname	Faculty	String	2
Sal	Faculty	double	3

2.1.8. Disk Space Management

The lowest level of software in the DBMS architecture, called the disk space manager, manages space on disk. Abstractly, the disk space manager supports the concept of a page as a unit of data, and provides commands to allocate or deallocate a page and read or writes a page. The size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be don in one disk I/O. It is often useful to allocate a sequence of pages as a contiguous sequence of blocks to hold data that is frequently accessed in sequential order. This capability is essential for exploiting the advantages of sequentially accessing disk blocks [RG 00].

2.2. The Data Definition Languages in DBMS

In this section we introduce the definition languages used by the different types of database management systems to describe the data. The relational database uses the data definition language (DDL) to describe the data. The object –oriented database management system (OODBMS) uses the object data language (ODL) to describe the data. In the following subparagraphs we introduce the DDL of the Structured Query Language (SQL) and the SDL, which is an object definition language, used by SHORE physical store [CWFH 94].

SQL's Data Definition Language

We define the schemas of the database by a data definition language (DDL). The DDL is a subset of the SQL. The DDL can be used to define the creation, deletion, and

modification of tables. For example, the CREATE TABLE statement is used to define a new table. The INSERT statement is used to insert the data of a single tuple into a table. In order to insert a tuple of data into a table PERSON, the user can issue the commands shown in Figure 2.

```
CREATE TABLE PERSON (sin: CHAR (20), name: CHAR (20),
                     address:  CHAR (20), PRIMARY KEY (sin))

INSERT
INTO PERSON (sin, name, address)
VALUES (6576,' John smith', '213 Bordeaux Montreal Quebec')
```

Figure 2: INSERT statement

In case of deleting a tuple, the user can issue the command indicated in Figure 3:

```
DELETE
FROM PERSON (sin, name, address)
WHERE sin =6576
```

Figure 3: DELETE statement

The UPDATE command can be used to change the values in the fields of the tuple.

The Integrity Constraints:

The integrity constraints (IC) prevent bad data to be entered in the database. The end user or the database administrator (DBA) defines his constraints against the instances of the data when he specifies the schema of the database. The DBMS is responsible to check the instances of data if it meets the constraints specified by the user. If the data does not meet

the constraints, the DBMS does not enter this data to the database and informs the user. The main integrity constraints are: The key constraints care about the fact that the minimal certain subset of the attributes of the tuple (we call it the primary key) should be unique. If the user tries to give an instance of a tuple with a key that has already exists in the table before, the DBMS stops this data and informs the user. In case that there are more than one subset that are unique and can be used to identify the tuple, we call them the candidate keys. The DBMS also should do the same check against them as with the primary key. The foreign key constraints are needed if a tuple has an attribute, which is a key in another table. The DBMS should make sure that the key constraints of that table have been considered. The system can do the foreign key checking only against the table that are already available in the system. In Figure 4,we give an example of expressing the key constraints using SQL:

```
CREATE TABLE Salary (sin: CHAR (20), name: CHAR (20),
                     address: CHAR (20) Wage: REAL,
                     PRIMARY KEY (sin),
          FORIGN KEY (name, address) REFERENCE WORKERS)
```

Figure 4: An Example of Key Constraints

The second way for the DBMS to prevent the writing of bad data instances in the database is by enforcing the general constraints. An example of general constraint is to require that a certain value lie in a certain range. If the value is given does not meet the assigned range, the DBMS does not accept it.

Shore's Object Definition Language

In this section, we introduce the second kind of the definition languages, the ODL. The ODL is supported by the OODBMS. The OODBMS is a programming language with support for persistent object [RG 00]. Since the OODBMS supports collection types, it provides a query language over collection. The Object Database Management Group (ODMG) has developed a query language, for querying collection types, which is Object Query Language (OQL). The ODL also support the user Abstract Data types (ADT).

The SHORE Data Language (SDL) embodies The Shore type system. SDL is the language in which Shore types are defined. SDL supports language-independent description of object-oriented data type. SDL is quite similar in nature to the Object Definition Language (ODL) proposal from the ODMG consortium, which is descended from OMG's Interface Description Language (IDL), a dialect of the RPC interface language used in OSF's Distributed Computing Environment (DCE) [CWFH 94]. In Shore type system, the interface type constructor constructs all the object instances. Interface types can have methods, primitive types of attributes or constructed types, and relationships. The set type constructors provided by Shore are numerations, structures, arrays, and sequences. Shore also provides bulk types, including sets, lists, figure and sequences. To group types together for name scoping and type management purposes, Shore provides modules. To provide an example of SDL representation of types, Figure 5 shows how one of the OO7 benchmark types can be represented [CWFH 94]:

Contents of the file oo7.sdl.

```
module oo7
{
  const long TypeSize = 10;
  enum  BenchmarkOp { Trav1, Trav2a, Trav2b, etc };  // forward
                                                     //declarations
  interface Connection;          // connector for atomic parts
  interface CompositePart;       // container for atomic parts
  interface PartIdSet;           // really just a C++ class
  interface AtomicPart {
          public:
          attribute long id;
          attribute char type[TypeSize];
          attribute long buildDate;
          attribute long x, y;
          attribute long docId;
          relationship bag to inverse from;
          relationship  bag from inverse to;
          attribute ref partOf;
          void swapXY();
          void toggleDate();
          void DoNothing() const;
          long traverse(in BenchmarkOp op, inout PartIdSet visitedIds)
          const;
          void init(in long ptId, in ref cp);
          void Delete();
  };// Connection, CompositePart, and other types go here
}
```

Figure 5: SDL representation of OO7 benchmark types

Shore language binding exists for C++ only. For example, the steps to create 007 bench mark application are:

- Write a description of the types in SDL (007.sdl file above)
- Use SDL type compiler to create type objects corresponding to the new types.
- Use language-specific tool to derive a set of class declarations and special-purpose function definition from type object. This step generates two files of

code 007.h and 007.c. The file 007.h is included in the C++ source files that supply the implementation of member functions and in the source files that manipulates instances. The object identification (OID) of the type object is compiled into these files and used to catch version mismatches at runtime. For more information, please see [SHOR].

2.3. Characteristics of STL Style

The Standard Template Library (STL) is a recent addition to the C++ language. It gives C++ a new level of abstraction. The standard gives us the ability to develop complex portable systems from scratch. All components of the STL are templates. The STL library provides the ability to use:

- string types;
- different sorting algorithms;
- numeric classes;
- classes for internationalization;
- data structures such as dynamic arrays, linked lists, and binary trees;
- input/output classes.

We can build many complete sophisticated modern systems with the majority of its building blocks obtained from the STL library. A large number of building blocks already exist with a complete implementation. This reduces the time needed for the implementation for many large systems where a great percentage of the code is easily imported from the STL. Matthew Austern wrote, " Computer programming is largely a matter of algorithms and data structures." [MAU99]. In STL, there are two main generic

types of building blocks, algorithms and the data structures. To support code reuse, flexibility and extensibility, the STL contains other generic building blocks such as iterators, functors and container adaptors that can be used to connect different blocks together to build large systems. Figure 6 shows STL components [J99]. The figure shows how the iterators are used as interfaces between the containers and the algorithms.

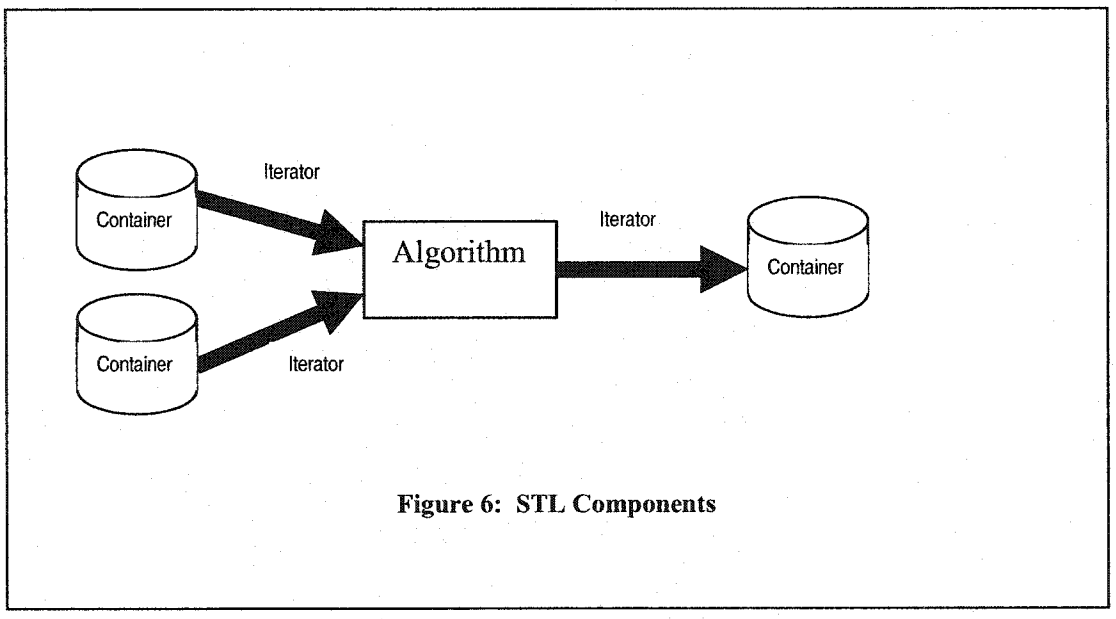


Figure 6: STL Components

Almost all classes and functions of the STL library are written as templates. Template function and classes are written for one or more types that are not specified yet. For example, the following code is a function that returns the minimum of two values. The type of the values should be passed explicitly or implicitly by template. Figure 7 shows how to pass the type of the values by template.

```
template <class T>
inline const T& less (const T& lhs , const T& rhs)
{
```

```
    return rhs < lhs ? rhs : lhs
}
```

Figure 7: Passing the type of the values by template.

2.3.1. Containers

Containers are used to process sets of data that have specific characteristics. There are different types of containers. Every container has its own properties that satisfy a specific application. Containers can be divided into two groups:

- Sequence containers: keep the elements in the same order the client put them.

  The STL contains three predefined sequence container classes:

  1. Vectors: are dynamic arrays where elements can be accessed randomly (directly) by using the element's index. Appending an element to the end of the vectors is fast. However insertion and deletion are slow since it takes time to move the following elements to make room for the inserted or removed element. For example, Figure 8 shows how, in the physical store system, we implemented class DataBase as a container of type vector of Table objects.

```
typedef std::vector < Table> DBtablesContainer;
DBtablesContainer Tables;                    //container of tables
```

Figure 8: An Example of Container

The type of the element saved in the vector is passed as a template parameter. Vector class has a push_back() function with a parameter of

type template element's type to append an element to the container. We want to mention here that strings in STL are implemented as a vector of characters.

2. Deques: are dynamic array where elements can be accessed randomly (directly) by using the element's index. Unlike vectors, deques can grow in two directions. Appending an element to the end or the beginning of a deque is fast. However insertion and deletion are slow since it takes time to move the elements to make room for the inserted or removed element. For example, we can use the following code, shown in figure 9, to declare a deque of type integer:

```
typedef std::deque < int > coll;
```

Figure 9: Declaring a Deque of Type Integer.

The type of the elements saved in the deque is passed as a template parameter. The deque class has push_back() and push_Front() functions with a parameter of type template element's type to append an element to the end of container and to add an element to the front of the container.

3. Lists: are doubly linked lists of elements where every element points to its predecessor and its successor. Lists do not have direct access. As a result, accessing the neighboring elements is very fast whereas accessing random element is slow. Besides that, insertion and deletion are very easy and fast.

- Associative containers: are collections of sorted elements where the position of the element depends on certain sorting criterion of the element's value. Associative containers are implemented as a binary tree where every element has one parent and two children. All ancestors to the left have lesser values whereas all ancestors to the right have greater values [SH 97]. The STL contains four predefined associative container classes:
    1. Sets: are collections of sorted elements according to the element's value. The duplicates, in sets, are not allowed.
    2. Multisets: are the same as sets but duplicates are allowed.
    3. Maps: are collections of sorted elements where the elements are a pair of (key/value). The position of the element depends on a sorting criterion, which is the key of the element. Duplicates key are not allowed.
    4. Multimaps: are the same as maps but duplicates are allowed.

The STL provide, in every container class, common operations such as initialization, size operations, comparisons and assignment and swap.

2.3.2. Container Adapters

The containers, in the C++ Standard Template Library, have a lot of characteristics such as associative, sequential and access type (random or sequential). These characteristics determine the functionality offered by the container to the clients. However, clients sometimes are interested in customizing these characteristics. Container adapters provide interfaces for the containers to customize their characteristics to fit some special needs

and to provide safe access for these special needs. Fortunately, STL provides three standard predefined container adapters that fit some special needs:

1. Stacks: are deque containers that are adapted by an interface to manage the elements by a last-in-first-out policy (LIFO). The relation between the deque container interface and the adopted stack container's core interface are shown in Figure 10 [J99].

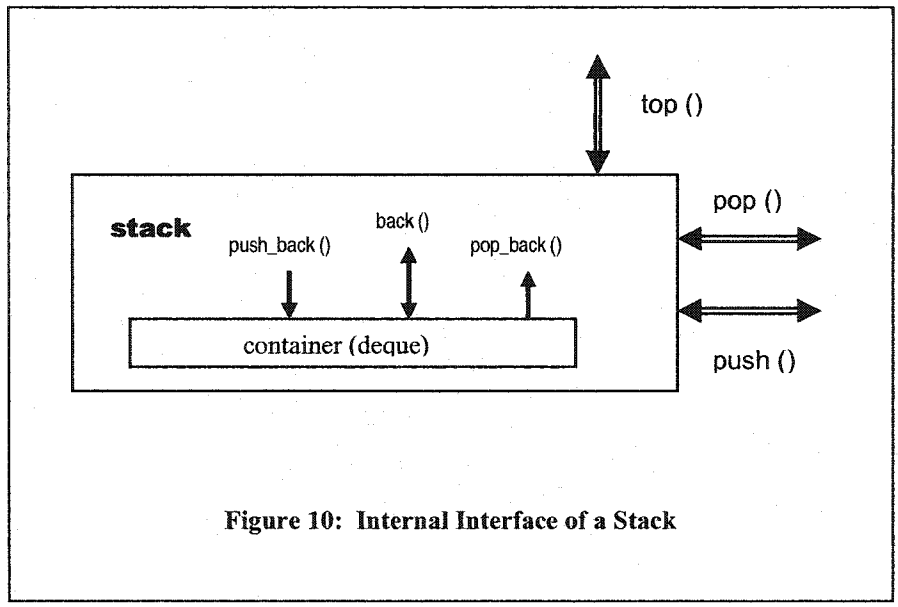


Figure 10: Internal Interface of a Stack

2. Queues: are deque containers that are adapted by an interface to manage the elements by First-in-last-out policy (FIFO).
3. Priority Queues: are queue containers. However, the elements are not managed by a FIFO policy. Rather, some priority algorithm manages the elements. In result, the push () function inserts an element into the queue, whereas top () and pop ()

access the next element, which is the element that has the highest priority. The default priority algorithm is operator < (less-than).

2.3.3. Iterators

An iterator class, in STL, can access the elements of a container without exposing its internal representation. It provides a uniform interface for traversing different containers. The design of an iterator class and its association with a container class (aggregate) are explained in [GHJV 97]. Figure 11 shows the structure of an iterator and the association with a container class (aggregate class) [GHJV 97]. The class iterator is implemented as a smart pointer and its interface is similar to a pointer's interface. The operations that are defined, in the iterator class, are:

1. Operator * is used to return the element of the current position. After dereferencing, we can use the operator -> to access the members of the element class directly.
2. Operator ++ and operator - - are used to step forward and backward.
3. Operator == and != are used to test if two iterators have the same position.
4. Operator = is used to assign an iterator.
5. begin() is used to return the iterator that represent the beginning element of the container.
6. end() is used to return an iterator that represents the end of the elements of the container.

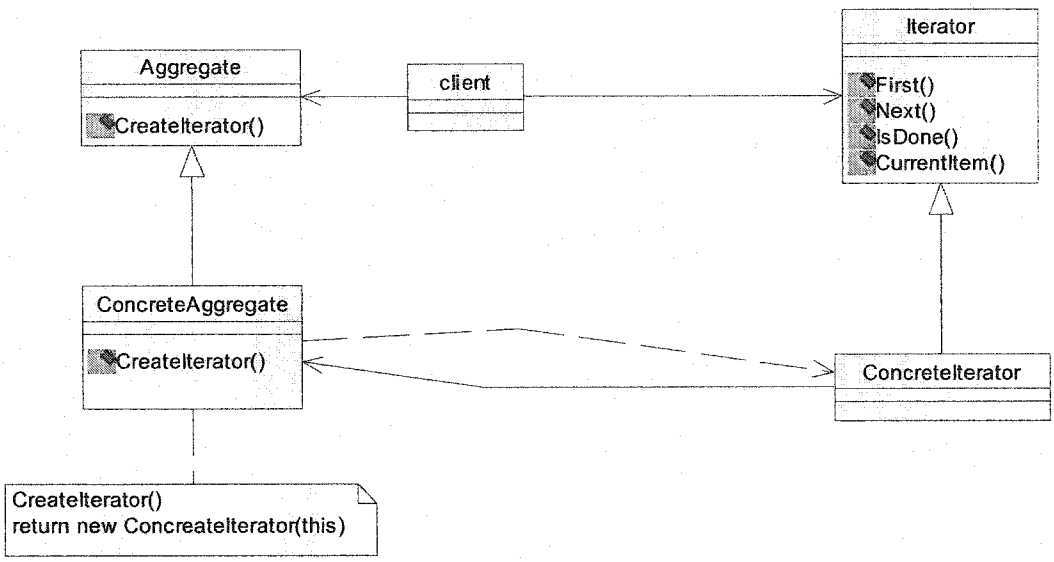


Figure 11: The Structure of the Iterator

There are different kinds of iterators according to their capabilities. We can use the capabilities of an iterator if and only if the characteristics of the related container allow a client to take advantages of such capabilities. For example, if a container has random access its iterators also can perform random access operations. The general abilities of the containers divide them into two categories:

1. Bidirectional iterators: have the ability to iterate in two directions: forward and backward. The iterators of the container classes such as list, set, multiset, map and multimap are bidirectional iterators.
2. Random access iterators: have the highest degree of freedom since they can move forward and backward and they allow direct access to an arbitrary element using the index operator [].

2.3.4. Algorithms

In the STL library, as we mentioned, we can use iterators as an interface between algorithms and containers. This fundamental concept, in STL, allows using the same algorithm on different containers. This concept of the generic algorithm contributes further to the concept of code reuse and increases the power and flexibility of the STL library. However this concept violates object-oriented design principles where the data and operations are unified. This concept has its own disadvantages. For example, the use of an algorithm is not intuitive and it is the responsibility of the users to see that use of the generic algorithms is useful. Predefined algorithms, in STL, are global functions that provide general fundamental services, such as searching, sorting, copying, reordering, modifying, and numeric processing. Algorithms access and process one or more ranges of elements of a container by passing the beginning and the end of the range as two separate arguments. For example the following code, shown in Figure 12, finds a certain value in a range of container's elements:

```
list<int> coll;
list<int>::iterator Position;
Position = find (coll.begin(), coll.end(),        // range
                 980);                             // value
```

Figure 12: Find Algorithm

2.3.5. Iterator Adapters

An iterator adapter is a class that has an iterator's interface and behaves like an iterator. The C++ standard template library provides several predefined special iterator adaptors:

1. Insert iterators to operate in insert mode rather than in overwrite mode.
2. Stream iterators to read and write to a stream.
3. Reverse iterators to switch the increment operator into decrement operator and vice versa.

This concept of iterator adapters contributes to code reuse and increases the power and flexibility of the STL library.

2.3.6. Function Objects

Function objects (or functors) are objects that are a generalization of function pointers simlar to the way of generalizing pointers into Iterators [RO00]. To define a function object, we have to define operator () in the class where we want to apply the function object technique. At compile time or run time, the function object is bound to a specific function. This concept of generalizing the function into functors adds more functionality and increases the power and flexibility of the STL library.

2.3.7. Allocators

Allocators are used to handle the allocation and deallocation of memory for the declared objects by the program written using the C++ standard template library. When we declare an object of any type, the C++ standard template library assumes that we intended to use

the default STL allocator implicitly if we did not explicitly pass another allocator implemented by the user. The default allocator is defined in the STL as shown in Figure 13.

```
namespace std {
  template <class T> class allocator;
  }
```

Figure 13: Default Allocator

The default allocator function, when we use STL components, calls the default new and delete operators. For example, the container vector has the format that is shown in Figure 14:

```
template < class T, class allocator = allocator <T> > class vector;
```

Figure 14: Container vector

If we want to declare an object container of type double, we simply write the code shown in Figure 15:

```
vector < double > myvector;
```

Figure 15: Vector of type double.

Now, when we insert into or delete from the container a double element, the allocation and deallocation of the memory is implemented by calling the default new and delete operators. We usually use the default allocator in our programs, however, if we have particular needs for managing the storage of the elements of the container we have to design and implement our own allocator and pass it as an argument to our objects. For example, in order to declare an STL vector object of type integer with a certain allocator, such as intAllocator, we pass this allocator as a template argument with the same type as the vector class. The following code, shown in Figure 16, creates a container of integers using the special allocator intAllocator:

```
std::vector < int, intAllocator < int > >
```

Figure 16: Special Allocator

Andrei Alexandrescu gives us a good example of using special allocator for small objects. The default free store allocator is often optimized for allocating large objects, not small objects. The default allocator is slow and brings important per-object memory overhead if it is used to allocate small objects. The small-object allocator class, in Loki library, overloaded the new and delete operator to optimize the allocation and deallocation of small objects and to eliminate the disadvantage of the default allocator. For more information please see [A02].

2.4. Techniques and Tools

In this section, we introduce the techniques and tools we used through our design and implementation of the physical store system. We discuss the Typelist that we used to implement the dataTuple class. Then we introduce the Boost library since we use it in implementing class myTuple. Then we introduce the mySQL parser that we use to compile the SQL user command that we used in physical store system and we finish by introducing POST++ library since we use the persistent allocator to implement class PhysicalStoreContainer.h.

2.4.1. TypeList

The theory part of this paragraph mostly follows [A02]. In our system, we use typelist to generate tuple code in the compile time. We can manipulate collection of types by using C++ tool Typelists. All the fundamental operations that list of values support are offered for types in C++ tool Typelists.

- Defining typelists

  There are only two types are held in the typelists. These types are Head and Tail. For example, we can declare a typelist of three elements holding the C++ types char, double and integer by writing the following C++ code shown in Figure 17:

```
typedef Typelist < char, Typelist < double, int > >
```

Figure 17: Declaring a typelist

- Linearizing Typelist Creation

The typelist library, namely the file Typelist.h in Loki library defines a plethora of macros that transform the recursion into simple enumeration, at the expense of tedious repetition. The repetition is implemented only once, in the library code, and it scales typelists to a large library-defined number. These macros liberalize the creation of typelist. The macros look as shown in Figure 18:

```
# define TYPELIST_1(T1)  Typelist<T1, NullTyp>
# define TYPELIST_2(T1, T2)  Typelist<T1, TYPELIST_1(T2)  >
# define TYPELIST_3(T1, T2, T3)  Typelist<T1, TYPELIST_2(T2, T3)  >
.....
# define TYPELIST_50(.....) .....
```

Figure 18: The Macros of typelist

- Calculating Length

In the Loki library, a recursive function is implemented to calculate the length of typelist.

- Indexed Access

In Loki library (namely in file Typelist.h), a recursive function is implemented to return the element of the needed index in the typelist.

- Searching Typelist

In Loki library (namely in file Typelist.h), a recursive function is implemented to return the index of the needed type element in the typelist.

- Appending to Typelists

Since modifying the typelists is not allowed because the types are passed to the typelists as template parameters, the solution is by "return by value " by creating a brand new typelist that contains the result. A function is implemented to return the output typelist that contains the input element type appended to the end of the list.

Table 3: The Compile-Time Algorithms Operating on Typelist

Primitive Name	Description
Length <TList>	Compute the length of Tlist.
TypeAt <Tlist, idx>	Return the type at a given position. If the index is greater than or equal to the length of Tlist, a compile time error occurs.
TypeAtNonStrict <Tlist, idx>	Return the type at a given position. If the index is greater than or equal to the length of Tlist, NullType is returned.
IndexOf <Tlist, T>	Returns the index of the first occurrence of type T in typelist Tlist.
Append <Tlist, T>	Appends a type or a typelist to Tlist.
Erase <Tlist, T>	Erases the first occurrence, if any, of T in Tlist
EraseAll <Tlist, T>	Erases all occurrence, if any, of T in Tlist
NoDuplicates <Tlist>	Eliminates all the duplicates from Tlist
Replace<Tlist, T,U >	Replace the first occurrences, if any, of T in Tlist with U.
ReplaceAll<Tlist, T, U>	Replace all the occurrences, if any, of T in Tlist with U.
MostDerived <Tlist, T>	Return the most derived type from T in Tlist.
DerivedToFront<Tlist>	Bring the most derived types to the front of Tlist.

Table 3 contains all the compile-time algorithms operating on typelist.

Class Generation with Typelists

In this section, we are going to define the fundamental constructs for code generation with typelists using the most powerful construct of C++ template template parameters. It is difficult to add a code for a type in a typelist since C++ language does not have compile-time iteration or recursive macros. The Loki library will help us add a code for a type in a typelist.

Generating Scattered Hierarchies:

The definition of GenScatterHierarchy, in the Loki library, is shown in Figure 19:

```
template <class Tlist, template <class> class Unit >
class GenScatterHierarchy;

template <class T1, class T2, template <class> class Unit >
class GenScatterHierarchy <Typelist <T1, T2>, Unit > : public
GenScatterHierarchy <T1, Unit >,
                                                    public
GenScatterHierarchy <T2, Unit >
{
  public :
        typedef Typelist<T1, T2> Tlist ;
        typedef GenScatterHierarchy <T1, Unit > LeftBase ;
        typedef GenScatterHierarchy <T2, Unit > RightBase ;
};


template <class AtomicType, template <class> class Unit >
class GenScatterHierarchy <Typelist <T1, T2>, Unit > : public
Unit<AtomicType>
```

Figure 19: The Definition of GenScatterHierarchy

```
{
  public :
        typedef Unit <  AtomicType > LeftBase ;

};


template < template <class> class Unit >
class GenScatterHierarchy <NullType, Unit >
{
};
```

Figure 19: Continued

We can pass a template class Unit to GenScatterHierarchy as its second argument. The GenScatterHierarchy class uses its template template parameter Unit just as it would have used any regular template class with one template parameter. The GenScatterHierarchy class passes the first template argument type to Unit in case that the first template type is atomic. The GenScatterHierarchy class also inherits from the resulting class Unit<T>. However, if the first argument of GenScatterHierarchy is a typelist Tlist, the class GenScatterHierarchy recourses to GenScatterHierarchy<Tlist::Head,Unit> and GenScatterHierarchy<Tlist::Tail,Unit>, and inherits both. The template<template<class>class Unit>Class GenScatterHierarchy< NullType, Unit > is empty class and it does nothing for the Null Type. In fact, when we instantiate an instance of class GenScatterHierarchy the object inherits Unit instantiated with every type in the typelist.

- **Generating Tuples Using GenScatterHierarchy**:

A tuple facility was first introduced by Jakko Jarvi and then refined by Jarvi and Powell [A02]. Tuple is a structure with unnamed fields. In Loki library, The Tuple template class is implemented as a GenScatterHierarchy. For example, the code is shown in Figure 20 for Person tuple:

```
template <class T>
struct Holder
{
    T value_;
};


typedef GenScatterHierarchy< TYPELIST_3 (int, std::string, std::string) ,
Holder > Person;
```

Figure 20: Person Tuple Using GenScatterHierarchy

The inheritance hierarchy generated by Person looks like Figure 21. This class hierarchy is generated by repeatedly instantiating a class template that is provided as a model. Then it collects all such generated classes in a single leaf class in our case, Person.

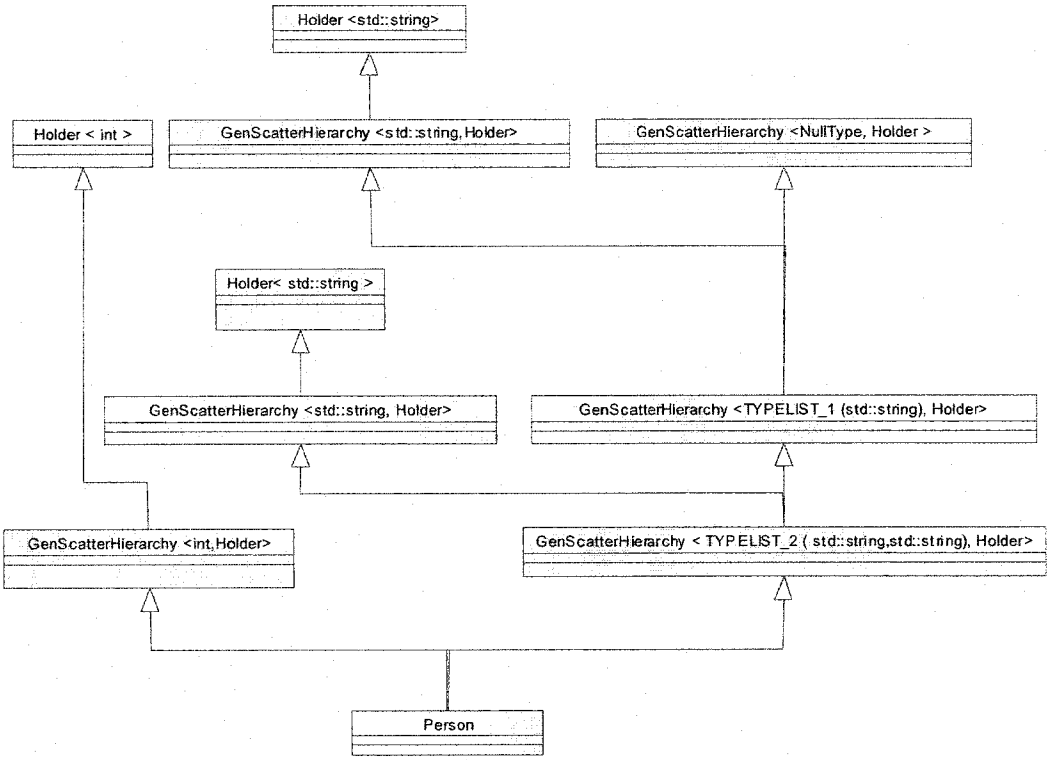


Figure 21: The Inheritance Structure of Person

- **The Tuple Template Class**

In Loki library, The Tuple template class Tuble.h is implemented similarly to GenScatterHierarchy. The Tuble.h class provides direct fields access where the Field access functions return references to the value_ member directly. Figure 22 shows an example of a tuple PERSON.

```
typedef Tuple < TYPELIST_3 (int, std::string, std::string) > PERSON;

PERSON p;

Field<0> (p) = 54654;
Field<1> (p) = "John Smith";
Field<2> (p) = "8115 Bordeaux, Montreal, Quebec";
```

Figure 22: An Example of a Tuple

Tuples are used also to return, from a function, number of different related values such as the person data or the multi dimensional coordinates of a point

2.4.2. Boost tuple Library

We use the tuple of Boost library in our recent implementation of class dataTuple in the physical store prototype. The Boost libraries are compatible with C++ standard Library. Some of Boost's libraries have already been proposed for inclusion in the C++ Standard committee's upcoming C++ Standard Library Technical Report [JJ 02].

Unlike tuple of Loki library, tuple in Boost is a fixed size collection of elements. A tuple is a data object containing elements of other objects. We use tuples to return number of types from a function, to assign number of values simultaneously and to group number of related types or objects together in a single entity. In the Boost library, the tuple type is an instantiation of the template where the template parameters defined the types of the tuple elements. Of course, we cannot use element types that do not have a copy constructor or default constructor. Figure 23 shows an example of constructing a tuple.

```
class myType {
        myType ();
                public:
        myType (std::string);
};

tuple < myType, int, std::string > (string ("jaba"), 8798798, "Duu")
```

Figure 23: Constructing of tuple

We can construct tuples also using the make_tuple function. This function is useful to hide the return type. Figure 24 shows an example of using make_tuple function.

```
tuple < int, int, double > myFunction (int a, int b) {
   return make_tuple (a*b, a+b, double(a)/double(b));

}
```

Figure 24: Using make_tuple Function

In Boost tuple, tuples can be copied and assigned. Tow tuples can be assigned as long as they are element-wise assignable. Figure 25 shows an example of assigning tuples.

```
tuple < char, int, const char(&) [3] > t1('a', 22, " HELLO" );
tuple < int, int, std::string > t2;
   t2 = t1;
```

Figure 25: Assignment of Tuples

The number of element in a tuple can be accessed as a compile-time constant using tuple_size function and the element type can be accessed using the template function tuple_element. Figure 26 shows the tuple_size function and tuple_element function.

```
tuple_size <tuple <  int, int, double > >::value;   //3
tuple_element <2, tuple <  int, int, double > >::type  // int
```

Figure 26: Tuple Functions

Beside that, we can apply on tuples relational operators such as == (equality), < (less than), > (greater than), >= (less than or equal) and >= (greater than or equal) if and only if such relational operators are defined for the corresponding elementary operator. The Streaming operator << is also defined in class tuple of Boost library. Since tuple access and constructors are small-inlined one-liners, a good compiler can eliminate any extra cost of using tuples compared to using hand written tuple like classes [BOOST].

2.4.3. mySQL Parser

In this section, we explain the parser in mySQL database system since we are using it in our system. mySQL is a relational database and it uses SQL commands [WA02]. mySQL is an open source code and it is written in C. Unlike the C++ programming language, the C language does not have type checking and type safety techniques. In mySQL database system, the parser is located in a parse directory.

The parser gets an SQL command, passes this command to the lexical analyzer to perform lexical analyzing. The algorithm that performs lexical analyzing is located in file lexyy.c in parse directory. After lexical analyzing, the SQL statement is parsed and the components of this command are saved in a parse tree data structure. The algorithm that performs parsing of the query is located in pars0parse.c file. The parser is implemented using the facilities available in Yet Another Compiler Compiler (YACC) [L 97]. The

facilities, available within the YACC system, lead to create a Parser for a language with a specified syntax. A parser generator is a program that takes as its input a specification of the syntax of a language in some form, and produces as its output a parse procedure for that language [L97]. The input of Yacc is usually a file with a .y suffix and the output is a file consisting of C source code for the parser (usually <filename>.tab.c). A Yacc specification file has the basic format Indicated in Figure 27.

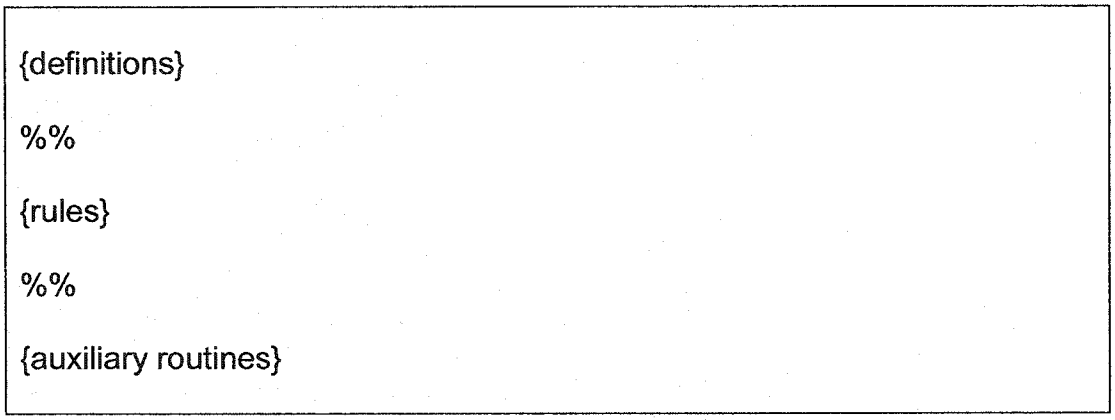


Figure 27: The Format of YAAC Specification File

The definition section contains information about the tokens, data types and grammar rules that Yacc needs to build the parser. It also includes any C code that must go directly into the output file at its beginning (primarily #include directives of other source code files).

The rules section contains grammars rules in a modified BNF form, together with actions in C code that are to be executed whenever the associated grammar rule is recognized. The Yacc file of mySQL includes the rules for all the SQL commands that used in physical store such as CREATE TABLE, CREATE DATABASE, INSERT, USE and LOAD INTO FILE. In figure 28, we show an example of SQL command:

```
CREATE TABLE tbl_name [(create_definition,...)]

CREATE TABLE Person (

id int(11) default NULL auto_increment,

s char(60) default NULL,

PRIMARY KEY (id)        ) ;


CREATE TABLE Test (

id INT NOT NULL,

last_name CHAR(30) NOT NULL,

first_name CHAR(30) NOT NULL,

PRIMARY KEY (id)    ) ;
```

Figure 28: Example of SQL Command

For more information please see [WA02].

2.4.4. POST++: Persistent Object Store for C++

POST++ is based on memory mapping mechanism and shadow pages transactions and it provides effective storage for the objects. POST++ supports several storages, storing objects with virtual functions, atomic data file update, provides high performance memory allocator and optional garbage collector for implicit deallocation of memory [POST]. POST++ correctly works with C++ classes using multiple inheritance and

pointers inside objects. Storage data file is mapped into virtual address of the application, allowing normal access to persistent objects. POST++ supports STL also. POST++ extracts the needed information about the persistent object classes by using special macros. The information is used to support garbage collection, relocation of references while loading and initialization of pointers to virtual tables.

Since C++ lacks the facilities to extract information about class format at runtime, we have to register our classes by using special macros provided by POST++. The user should include macro CLASS_INFO (NAME, FIELD_LIST) in the definition of every class that will be saved and retrieved by POST++ facilities. The NAME corresponds to the name of the class and FIELD_LIST describes references field of the class. There are three macros defined in file classinfo.h for describing references: REF (X) to describe single reference field, REFS (X) to describe array of fixed number of references, and VREF (X) to describe array of variable number of references. Macro CLASSINFO defines default constructor to declare class descriptor for our classes. In Figure 29, we introduce a simple example of using POST++.

```
class myclass {
 protected:
   myclass *    next;
string * arr[10];
   int     x;
     public:
   CLASSINFO(myclass, REF(next) REFS(arr));
   myclass  (int x);
};

REGISTER(1, myclass);

main() {
   storage my_storage("myclass.odb");
   if (my_storage.open()) {
      my_root_class* root = (my_root_class*)my_storage.get_root_object();
        if (root == NULL) {
           root = new_in(my_storage, my_root)("some parameters for root");
        }
        ...
      int n_arr = ...;
        size_t varying_size = (n_arr -1)*sizeof(string*);
        // We should subtract 1 from n_arr, because one element is already
        // present in fixed part of class.
      myclass* fp = new (myclass:self_class, my_storage, varying_size)
myclass(x);
        ...
        my_storage.close();
   }
}
```

Figure 29: An Example of Using POST++

In order to make the allocation and deallocation very efficient, POST++ provides special memory allocators with two strategies: for allocating small object and large objects. The memory manager of POST++ uses first fit, random position strategy to maintain list of free pages. In POST++ mark and sweep garbage collection scheme is used. The garbage collector first marks all objects accessible from the special root object available in the

storage. Then it collects all the objects that are not marked during first stage of garbage collection.

All classes of persistent objects in POST++ should be derived from the object class defined in object.h.

A programmer needs some kind of root object to access an object in persistent storage. Each other object can be accessed by normal C pointers starting from the root object.

We can use several storages in our application simultaneously [POST]. A storage constructor takes a path to storage file as mandatory argument whereas the maximum size of the file and the maximum number of locked objects in transaction mode are optional since they have default values.

The data consistency, in POST++, is supported by two different strategies. The first strategy based on transaction mechanism using shadow pages to provide storage recovery. The second strategy based on copy on write mechanism where the original file is not affected and a copy of the page is created if any modification takes place in the page mapped on the file. The copy of page is allocated from system swap and has read-write access. A File is updated only by explicit call of storage::flush() method. This method writes data to temporary file and then renames this file to the original one.

In order to make STL objects persistent objects, POST++ redefined operators new and delete in special allocator designed for STL objects.

POST++ can store and retrieve STL classes from the POST++ storage. The interface to STL provided by POST++ does not require any changes in STL classes. POST++ is not able to perform garbage collection and reference adjustment since STL classes do not contain type descriptors. POST++ should be always mapped to the same virtual address

in case of using STL interface. Figure 30 shows the UML class diagram of the POST++ persistent allocator that used to store and retrieve STL classes from the POST++ storage. In the diagram, we see the template class post_alloc that contains the implementation of the allocator. This allocator should be passed instead of the default allocator of STL library when we declare a persistent container (see section 2.3.7). In class post_alloc, there are two important functions: allocate and deallocate. Allocate function checks if the object is available in the storage, if so it retrieves the object otherwise it allocates new one after getting information about its size of the object. The deallocate function deletes the object from the storage. Class Storage is the interface for saving and retrieving objects in the physical files. Class object is the base class for all the objects we need to save and retrieve from the persistent storage since all should be derived from the object class. Class File is responsible about the low level operation of reading and writing to the physical file. Class class_descriptor is a supporting class to hold information about the classes we need to transfer their objects from and into the persistent storage. Class simple_mutex is used to lock and unlock the storage to achieve concurrency.

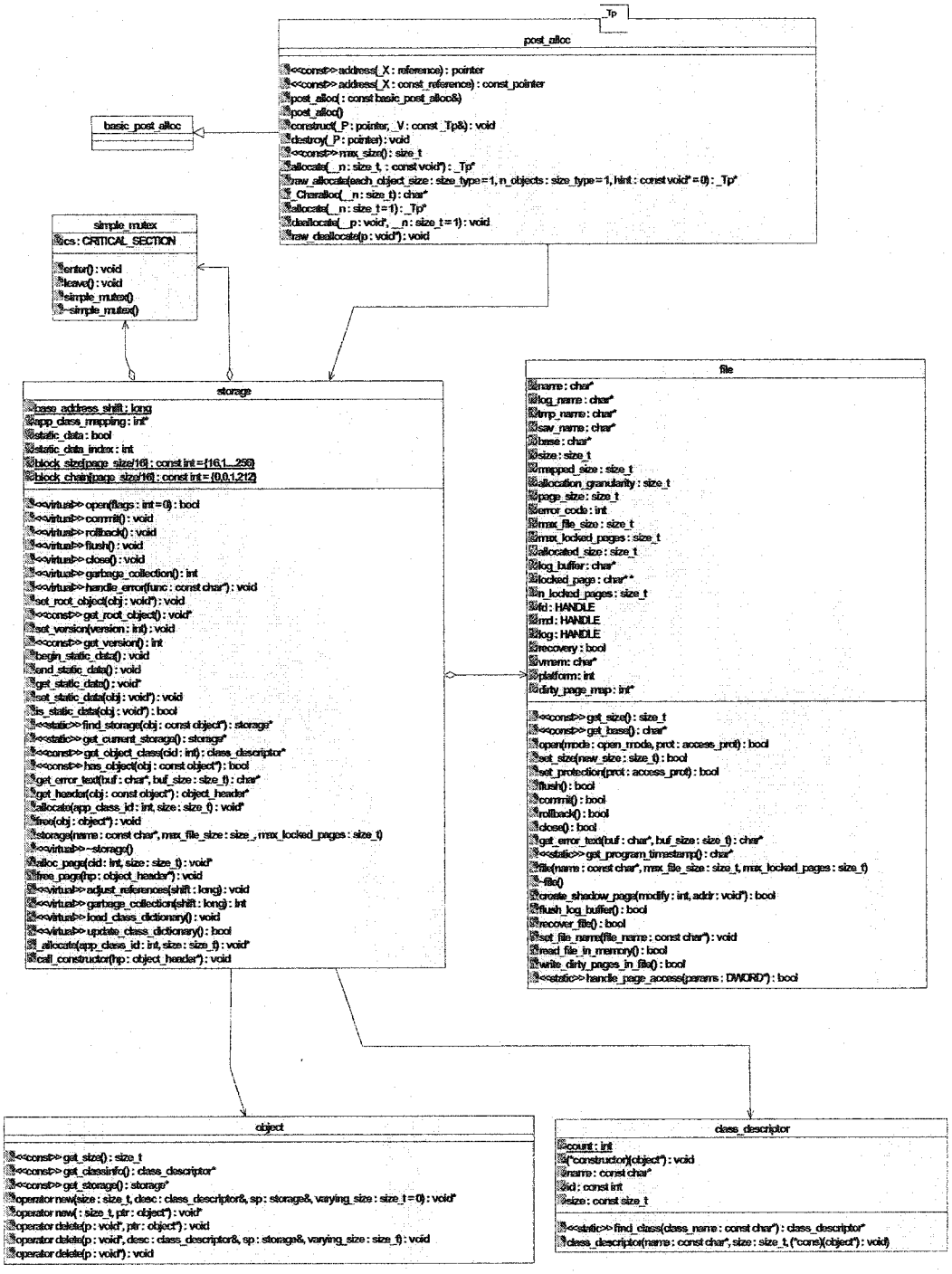


Figure 30: Classes' Diagram of POST++ Persistent Allocator for STL Classes

2.5. Design Patterns

In this section, we introduce the design patterns that we use in the design of the physical store system.

2.5.1. Factory Method

The Factory Method design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory method is also known as virtual constructor [GHJV 95]. The standard structure of factory method is shown in Figure 31 [GHJV 95].

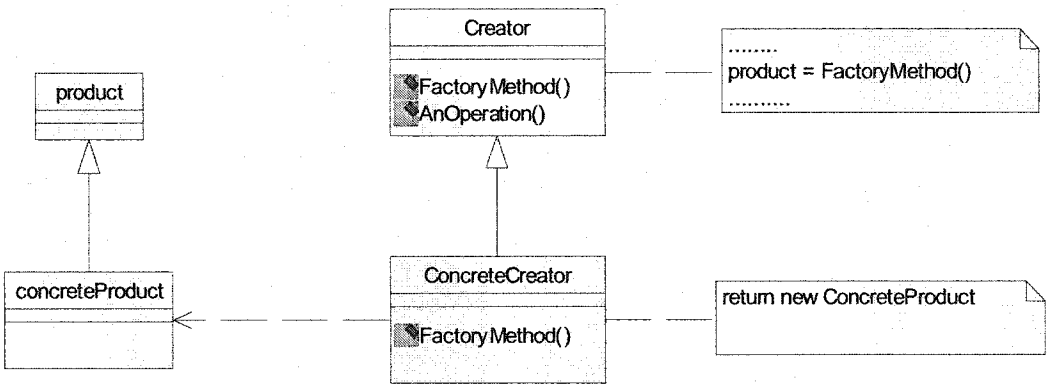


Figure 31: The Structure of Factory Method

Product is the interface of the objects the factory method creates. ConcreteProduct implements the product interface. Creator declares the factory method and may call the factory method to create a Product object. ConcreteCreator returns an instance of ConcreteProduct by overriding the factory method.

In the physical store system, the users and the clients can decide the type of buffer that should be used when the system runs their applications. They can choose the buffer that

fits their application needs. A default buffer is also available. For example, the class table does not know what kind of buffer, fixed length buffer or variable length buffer, to use so it defers the decision to the run time when the user or the client class identifies the type of buffer that fits its need. The class IOBufferCreator plays the role of Creator class. The Buffer abstract class plays the role of Product and fixedFieldBuffer plays the role of concreteProduct in the factory method design pattern.

2.5.2. Proxy

The Proxy design pattern provides a surrogate or placeholder for another object to control access to it. Figure 32 shows the standard structure of Proxy design pattern [GHJV 95].

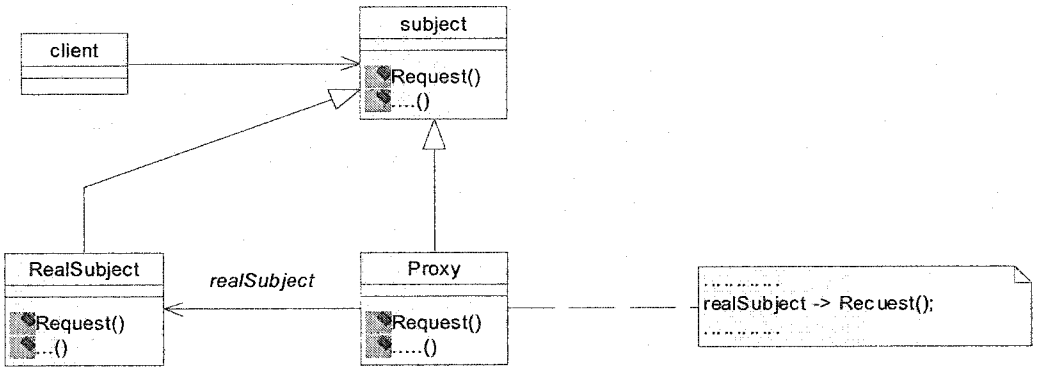


Figure 32: Proxy Structure

Proxy accesses the real subject by maintaining a reference to the real subject and controls access to the real subject. Subject is the interface of real subject RealSubject. RealSubject is the real object that proxy represents.

2.5.3. Generic Smart Pointers

The theory part of this section mostly follows [A02]. Smart pointers are widely used in implementing different kind of systems. They simulate simple pointer and combine many syntactic and semantic issues. Smart pointers provide to the client useful tasks such as memory management and locking. In smart pointer class we overload the operator * and the operator -> to give the smart pointer pointer-like syntax and semantics. The example, in Figure 33, shows how we use * and -> operators of generic smart pointer in our code:

```
template <class T>
class SmartPointer
{
  public:
    explicit SmartPointer (T* pointee) : pointee_ (pointee) ;
    SmartPointer & operator = (const SmartPointer & other);
    ~ SmartPointer();
    T & operator *() const;
    T * operator -> const;
  private:
    T* pointee_;
  ..................

}


SmartPointer <myClass > Sp (new myClass);
Sp-> doSomething();          // doSomthing() is a member function in myClass
(*Sp). doAnotherthing();     // doAnotherthing() is a member function in
                             //myClass
```

Figure 33: Using Smart Pointers

The example shows that replacing pointer definitions with smart pointer definitions does not increase our application's code. Unlike simple pointer, smart pointers have value semantics, can be copied and assigned to and they eliminate the resource lake problem by

managing the ownership. The vital functions of the smart pointers are constructing, copying, and destroying. There is number of strategies to handle the smart pointer ownership:

- Deep copy strategy which always copies the pointee object whenever the smart pointer is copied. Therefore, every smart pointer has its own pointee and the smart pointer's destructor can safely delete the pointee object.
- Copy on write strategy (COW) clones the pointee object at first attempt of modification.
- Reference-counting strategy and Reference linking strategy delete the smart pointer only if the number of smart pointers that point to the object equal zero.
- Destructive copy strategy destroys the object being copied.

In smart pointers the multithreading can be controlled by two levels:

1. Multithreading at pointee object level: where we do not need to change any data inside the object and it is enough to just lock the object. This is possible by having the smart pointer return a proxy object. The proxy's constructor locks the pointee object whereas the destructor unlocks the pointee. In Figure 34, we present code that illustrates multithreading at pointee object level [A02].

```
Class myClass {
                    ....
                    Void lock ();
                    Void unlock;
                     .....
};

template <class T>
class LockingProxy {
          public:
            LockingProxy(T* pObj) : pointee_ (pObj)
                              {        pointee_ ->Lock(); }
           ~LockingProxy(T* pObj) : pointee_ (pObj)
                              {     pointee_ ->Unlock();   }
         private:
            LockingProxy & operator = (const LockingProxy &);
           T* pointee_;
};
```

Figure 34: Multithreading at Pointee Object Level

2. Multithreading at the Bookkeeping Data level: where we not only lock the object but we also need to change some shared data inside the object such as the reference count and reference link.

3. Physical Store Specification

In this chapter, we introduce the specification of the physical store. It includes the specification of the interface and the data definition language that should be provided by our project, the main tasks and function provided for the users and for the other layers and subframeworks of the Know-It-All framework. The physical store should support the separation between the index framework and the physical store system. The index framework and the physical store system communicate through the data reference and the keys. The separation between the physical data and the indexes allows us to build a number of indexes to the same data. On the other hand the change in the data does not affect the index if the keys remain the same. This separation is also important in the reusable object oriented design. The physical store should be modern sophisticated software that uses the concept and the technology of the reusable object–oriented design. The system should be implemented in a coding language that provides portability to the project. The user should define its data in text format using a friendly data definition language like the structural query language (SQL).

3.1. The Main Functions of the Physical Store

In this paragraph, we introduce the main functions that the physical store should provide to the different clients of the KIA project and the other layers.

The physical store should provide a service to save tuples of data in a persistent store and returns DataRef object. The DataRef object can be passed to the index layer. The client

can use the data reference object later to retrieve the same tuple of data from the persistent store.

The physical store should provide a service to retrieve from the persistent store a tuple of data by dereferencing the DataRef object, which should be implemented using the smart pointer technology. The smart pointer also should provide an efficient way to minimize the number of the operating system heavy weight processes of input/output by postponing the heavy weight processes of I/O operations until we actually need to use it.

The physical store should provide a service to derive the values of the primary and the secondary keys from the data of the tuple and to pass these keys to the index layer so that the index layer can build up the needed indexes for the current table using the pairs of data reference and keys.

The physical store should provide an efficient algorithm to execute the bulk-load since the bulk load is one of the main goals in the KIA project.

3.2. The Specification of the Data Definition Language

In this section, we introduce the specification of the data definition language in the physical store. The physical store should provide friendly user interface. The user should have the ability to issue his commands in text file mode. The following are the different commands that the user can issue and the response of the physical store system to them.

In case of SQL command CREATE DATABASE: It creates a database with the given name. Databases are directories containing files that correspond to tables in the database. Because there are no tables in a database when it is initially created, the CREATE DATABASE statement only creates a directory under the KIA data directory. This

command should have a parameter with the name of database. The name of the database should be unique. If the used name is available already in the system, the physical store issues an error message. This database will be the default database and all the other command should be executed against it. The default database can be changed if the user issues a new CREATE DATABASE command or he explicitly issues the command USE to switch the current database.

In case of SQL command USE: The system should substitute the current default database with the new database. This command should have a parameter with the name of database that should be the default. The system considers that all the following user commands will be executed against the new database.

In case of SQL command CREATE TABLE: It creates C++ classes for the tables according to the format of the persistent store dynamically. The command CREATE TABLE has parameters to specify the name of the database where the table belongs and the name of the table. The command CREATE TABLE has parameters to specify the attributes, dependencies, restrictions and the keys. The command also has parameters to define the index and its parameters such as the key is used to build the index. At least one key should be defined as a key for building an index since the physical store should build a container of pairs of DataRef and key.

In case of SQL command LOAD DATA INFILE: It reads a file of data (text format) into the specified table. This command should specify the name of the table. The command LOAD DATA INFILE has parameters to specify the name of the text file of data and its path of access. The physical store takes the first number of fields that meet the same number of fields in the metadata of this table. If the number is less than the number of

fields specified in the metadata the system issues an error message. The system processes the data according to the types specified in the metadata of the table.

The command LOAD DATA INFILE has options to specify the end of the line and the fields' separations. The user can decide another parameter values such as the size of the buffer, whether it is necessary to apply all the database integrity constrains, such as the primary key, the secondary key and the foreign key. The checking of the integrity constraints plays a major role in the implementation time.

In case of SQL command INSERT: The physical store should insert a new row of data into an existing table. This command shoud specify the name of the table. The row of data should be passed as a parameter of this command. The command INSERT has options to specify the end of the line and the fields' separations. The physical store takes the first number of fields that meet the same number of fields in the metadata of this table. If the number is less than the number of fields specified in the metadata the system issues an error message. The system processes the row data according to the types specified in the metadata of the table.

3.2.1. Specifications Concerning the Index Layer

In most of the existing relational databases, there is no separation between index layer and physical layer. In the physical store, the separation between the physical data and the indexes allows us to build a number of indexes to the same data whereas the change in the data does not affect the indexes if the keys remain the same. In command CREATE TABLE, there are parameters to define the index and its parameters such as the keys used

to build the index. The physical store has to build a container of pairs of key and DataRef and to pass the iterator of the container to the index layer to build the indexes.

In case of SQL command CREATE INDEX: It adds a new index to an existing table. This command should have parameters to specify the type of the index and the key is used to build the index.

3.2.2. The Main Goals of Designing the Bulk-Loading

Efficient bulk loading is an issue when loading a huge amount of data into a table and its index. When creating a new index we want to provide it as fast as possible to the users and we want to gain the best possible performance. The ability to store vast data sets efficiently can have a dramatic effect on the overall cost associated with the maintenance of a data warehouse application. The source data is usually provided in a portable format, e.g., ASCII flat files, XML, Excel-Table, etc., but not in binary format. The main goals, which should be achieved by bulk loading techniques, are the following:

1. Minimize random disk accesses that cause an increase in the input/output processes. We can achieve that by adopting a good sorting strategy.
2. Minimize the heavy weight processes of the operating system that are caused by input/output processes to the secondary store. We can achieve that by adopting good paging and buffering strategies.
3. Minimize CPU load by adopting an efficient algorithm that relies on the concurrency strategy of the operating system [SGG00].

The main cost for loading arises from accesses to secondary storage. Strictly speaking, the loading process has a very high cost since it mostly consists of heavy-weight operating system input/output processes. Therefore, it is essential to minimize the number

of input /output processes. Mainly, in our algorithm, we have to avoid random accesses to the secondary storage wherever it is possible. However, the technology of caching that uses random access memory (RAM) between the main memory and the secondary storage supports paging from the secondary store to the main memory. Recent operating systems have very efficient algorithms that support the caching techniques of the linear secondary storage accesses. On the other hand, the CPU load also should be recognized as a critical factor for the loading process and should be minimized as well. With respect to query performance it is important to provide good clustering, since this reduces random disk accesses for range queries. There are two types of clustering, namely tuple and page clustering [RG00]. For bulk-loading it is possible to achieve both, in other words not only the tuples within one page should be clustered, but also the pages should be clustered according to the criterion of the index used. Additionally, a good degree of page filling should be achieved in order to decrease the number of page splits when we process range queries against a database.

3.3. The Application Programming Interface of the Physical Store

In this section, we introduce the different application programming interfaces of the physical store system and the context of using the system. The name of the routines, their parameters and types and the results of the execution are also described. The API defines how programmers and users utilize the routines, protocols and tools of the physical store. Since depending on the user, the physical store system is manipulated in different ways. The expert developer designs the system that provides the needed services whereas the developer sets the needed database parameters and adjusts the system to fit some specific

services such as bulk-loading. The database administrator (DBA) designs the schemes, the physical tables and indexes and issue command such as CREATE TABLE, CREATE DATABASE, whereas the user of the system issues commands against the data such INSERT, USE and LOAD DATA INFILE.

1. API: Create database

Routine name: createDatabase ().

Parameters: Parse tree structure containing the SQL command information such as the name of the database.

Result: The physical store system creates a directory with the specified name. Later, the related files to the assigned database saved under this directory so that the data can be found easily.

Description: The user specifies the name of the database and designs the schema of the database by specifying the names of the tables, the needed indexes, the primary and secondary keys, the foreign keys, the constraints and the attributes. Then he repeats the Create table routine for every table in the schema. All the metadata of the database are initialized using initialize metadata routine.

2. API: Create table

Routine name: createTable ().

Parameters: Parse tree structure containing the SQL command information.

Result: The physical store system creates a file with the specified name under the directory of the current database. The field's names and their types, the keys and constraints are saved in the system dictionary.

Description: The user specifying the names of the tables, the needed indexes, the primary and secondary keys, the foreign keys, the constraints and the attributes. If the default database is not the needed database, the USE routine is used to make the default database is the needed database. The result of this routine is that the metadata of this table is added to the metadata of the database using initialize metadata routine and the create class dynamically routine is run to create a corresponding class of the table's field types.

3. API: USE

Routine name: use ().

Parameters: Parse tree structure containing the SQL command information.

Result: The physical store system substitutes the current database with the specified one.

Description: The user specifies the needed database. The system makes the needed database as the default database.

4. API: Enter parameters

Routine name: enterParameters ().

Parameters: The size of the buffer: int, the type of the buffer: string, page utilization: int, checkForeignKey: int, checkPrimaryKey: int. All the parameters have default values.

Result: The physical store system substitutes the current values of the parameters with the specified values.

Description: The user can specify the parameter values such as the size and the type of the buffer, the index page utilization and whether it is necessary to apply all the database integrity constraints, such as the primary key, the secondary key and the foreign key. The constraints checks play a major role in the implementation time. The type of the buffer also plays role in decreasing the time of execution.

5. API: load data into file

Routine name: loadDataIntoFile ().

Parameters: Parse tree structure contains the SQL command information.

Result: The physical store system inserts into current database the specified data file.

Description: The user specifies the name of the data file. Then the Bulk load routine is run if the tables do not have indexes otherwise the incremental load routine is run.

6. API: Create index

Routine name: createIndex ().

Parameters: Parse tree structure contains the SQL command information.

Result: Physical store system invokes the index layer to create new index.

Description: The user specifies a new index to existing table. If the key of the index does not exist in the container of keys of the table, it creates dynamically a class key using create class dynamically routine and it adds this class to the keys container of the table. Then it run bulk-load algorithm to derive the values of the new key and pass it to the index layer to build the new index.

7. API: Dereference a DataRef

Routine name: retrievData ().

Parameters: key.

Result: The physical store system retrieves the needed tuple from the secondary storage.

Description: The client needs to retrieve some data by defining their keys. The system will find the data references of the needed data using the index layer. The data references will be dereferenced by the physical store system to retrieve the needed data from the secondary storage.

8. API: Derive the key

Routine name: keyDerive ().

Parameters: tuple: dataTuple.

Result: the value of the key corresponding to the metadata of the key is returned.

9. API: Insert

Routine name: Insert ().

Parameters: Parse tree structure containing the SQL command information.

Result: The physical store system inserts into current database the specified data.

Description: For an insertion, the physical store saves the data in a corresponding object of class table, it derives the key and saves the data in the corresponding file and adds the pairs of data reference and key to the container and passes its iterator to the index layer. The index layer runs its search routine to find a suitable place in the index. The position is then used to insert the pairs of data reference and key, and the index is adjusted if necessary to reflect the changes.

10. API: Initial load

Routine name: initialLoad ().

Parameters: Parse tree structure containing the SQL command information.

Result: The physical store system inserts into current database the specified data file.

Description: The physical store reads the file, save the data in the tables' files, builds the data reference keys container and passes the containers to the index layer. Then the index bulk load algorithm builds new indexes.

11. API: Incremental load

Routine name: incrementalLoad ().

Parameters: Parse tree structure contains the SQL command information.

Result: The physical store system inserts into current database the specified data file.

Description: The physical store reads the file, save the data in the tables' files, builds the data reference keys container and passes the iterator of the containers to the index layer. Then the index incremental load routine is run to insert the pairs of data references and keys into the existing indexes.

3.4. Physical Store Data Model

Figure 35 shows the data model of the physical store. The data components of the physical store, mainly, consist of:

Index: the data components of the index are number of primary and secondary indices. Usually, the same data set is used to build all the indices. The primary and the secondary indices, in their turns, consist of a number of data references and keys. The key consists of number of attributes. The indices files do not contain data of the database tables and records. The indices files contain pairs of key and data reference. We can change the physical data format without affecting the indices since the data is separated from the indices. This separation, in the other hand, gives the ability of building a number of indices for the same data. The indices are used to access the data using the different searching keys. Querying the index using a certain key leads to finding way through the index down to the index leaf level, where the index data is stored. The responsibility of the index is ended when it provides the data reference. Then the physical store system uses the data reference to access the tuple in the data file by applying one of its accessing mechanisms.

Data: The data component, in its turn, consists of number of tables and one catalogue. Every table and catalogue, in its turn, contains tuples whereas the tuple's data components are attributes. The catalogue maintains information about the physical store and its tables, indexes, views and the users authorities. Physical store system updates the information in the catalogue and uses it to find the needed information about the needed database.

System log data: The log files contain information about how the processes and transactions executed in the physical store system. The data logs are used for achieving consistency and data recovery in the system.

Metadata: Describes the data in physical stores. It contains information about the schemas of the database and its constraints.

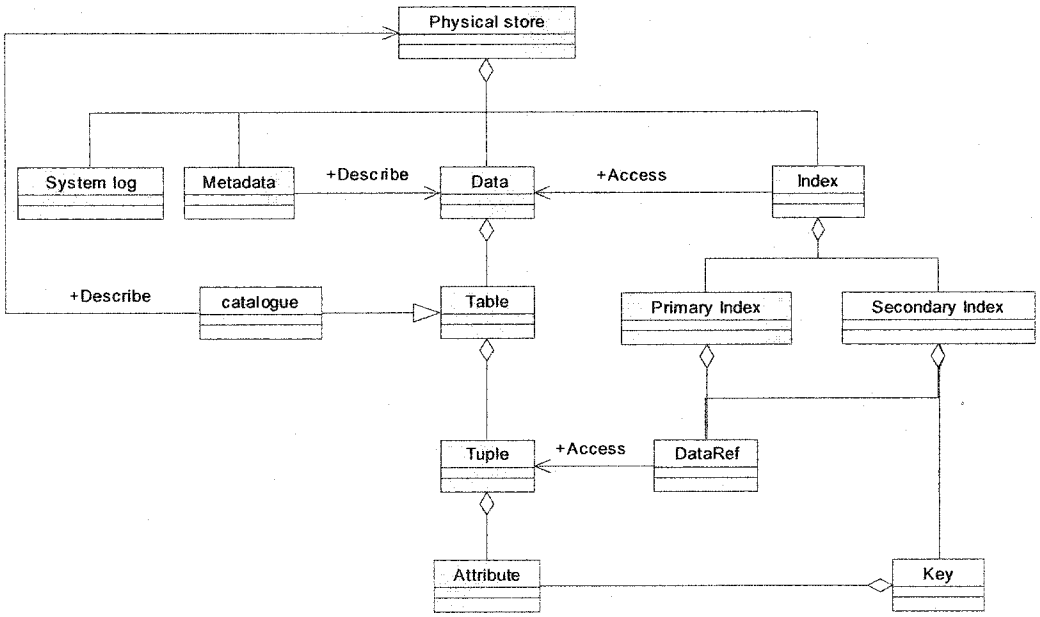


Figure 35: Physical Store Data Model

4. The Physical Store System Design

In this chapter, we introduce the architecture of the physical store and we present an object-oriented design for the physical store system. We discuss the structural aspects of this project, the overall structure, and the detailed design. We introduce the implementation of some classes of the system and the concepts that we use in our implementation. The scenario of how the client interacts with the components of the system is also presented.

4.1. The Module Architecture of the Physical Store

In this section, we introduce the next step in our software process. We derive the module architecture of the physical store directly from the assigned specifications and we describe its architecture. We also provide an abstract description of the organizational and structural decisions that are evident in the physical store system. The structural issues usually include gross organization and global control structure, protocols for communication, synchronization, and data access, assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance, and selection among design alternatives. This is the software architecture level of design [GS93]. In the physical store module architecture, we describe the decomposition of the software and its organization into layers. The subsystems are also decomposed into modules; the use-dependencies are also described. Figure 36 depicts the physical store module architecture. In the architecture representation, we use UML notation [JGJ 97].

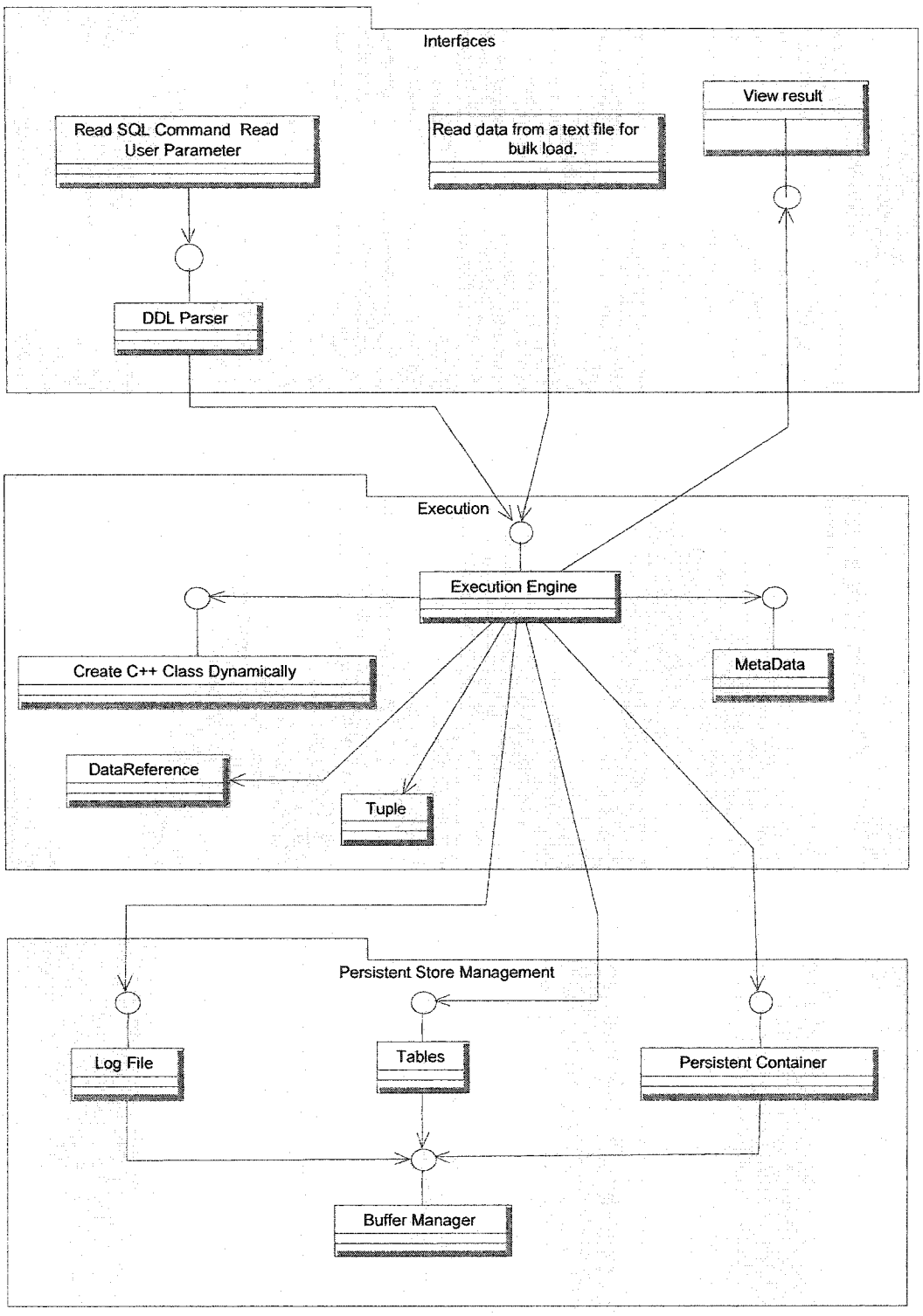


Figure 36: The Physical Store Module Architecture

The main components of the physical store architecture are:

1. **Interfaces subsystem**

In the physical store, the query interface of the Read SQL command Read User Parameter module, allows the user to issue SQL statements to the physical store and the View result module to view the result. The Read SQL command Read User Parameter module allows the user also to enter his parameters such as the type and the size of the buffer and the type of constraints he wants to enforce in case of bulk load since these constraints play major role in the performance of the system. The Read data from a text file for bulk-load module provides an interface to read data from an external text file since the data of the bulk load is saved in a text file. The command should be issued in a text format. The user can also specify the parameter values such as the index page utilization and whether it is necessary to apply all the database integrity constraints, such as the primary key, the secondary key and the foreign key. The constraint checks play a major role at run time. The type of the buffer also plays role in decreasing execution time.

The DDL parser module

In this stage, the objective of the data definition language parser is to create a parse tree structure based on the SQL command line. The components of the following stages of the processing can easily understand the user's query by traversing the parse tree. In this stage, the parser should check the SQL syntax and check the semantics of the command to determine if the command is valid. If it is a valid command, then the command

progresses down the pipeline. If not, then the command does not proceed. The physical store should notify the client that a query processing error has accursed.

The execution engine uses the DDL parse tree, as obtained from the DDL parser, to run the corresponding algorithms.

2. The Execution subsystem

The execution engine is the heart of the physical store system. It is responsible for declaring and initializing the objects of the system and passing the control between the subsystems and the modules. It receives the SQL command from the DDL parser in a parse tree structure and it uses it to instantiate the metadata and to create the tuple and key classes.

The DataReference module

The DataReference module is responsible for saving and retrieving the data. We use the smart pointer technology to achieve a number of tasks such as memory management, memory locking, and to manage the lifetime of the objects. The main purposes of the DataReference module are to hide all the functions required to save the data in a file and retrieve the data from a file and to retrieve the data from a file only when we really need these data to decrease the I/O cost.

Create C++ class Dynamically module

This module is responsible for creating the C++ classes for tuples and keys by using typelists. It creates an object dynamically of type typelists then creates dynamically an object of type tuple with the same types as these of the typelist. The Tuple module is the interface that uses the Create C++ Class Dynamically module.

The Execution Engine module

In the physical store the execution engine consists of the following procedures:

- Procedure to instantiate the metadata classes with the information available in the parse tree.
- Procedure to create dynamically C++ classes for the data in the tuple and their keys using typelist technology.
- Procedure to read the data from a text file in case of implementing bulk load algorithm.
- Procedure to derive the key values from the dataTuple.
- Procedure to save the data in the persistent store and to create an object of type DataRef for every tuple.
- Procedure to pass the pairs of (DataRef, key) to the index layer. The index layer can build the needed indexes using these pairs.
- Procedure to present the results of the processing to the user.
- Procedure to access the data using the DataRef.
- Procedure to reterieve data frome persistant store and using C++ template typecast to insure type safety.

Metadata module

Metadata describes the data in physical stores. It contains information about the schema of the database and its constraints. The information about the metadata is received from the user by SQL command. After parsing the SQL command by the DLL parser and saving the information in a parse tree, the metadata is extracted from the parse tree and saved in the metadata module by the execution engine.

3. Persistent store management subsystem

The persistent store management subsystem consists of Buffer Manager module that controls and manages the buffer, the Persistent Container module for saving the indexes, the Tables modules for saving the databases and the Log File module for saving the system logs. In addition, it provides the to and retrieval of the data using the data reference or the persistent allocator of the container.

The Persistent Container module

The Persistent Container module is responsible for storing the pair (DataRef, key) that the physical store provides to the index layer to create the needed indices for the tables. The bulk-load function needs a persistent store to temporarily save the pairs (DataRef, key) of large amount of sets of tuples in order to apply on these pairs external sorting by the values of their keys. It is also responsible for saving the data of the index layer. This maintains the compatibility with the design and implementation of the index layer.

4.2. The High-Level Design

In this section, we introduce the high level design of the physical store system. In this level of design, we convert the architecture of the physical store into classes. Figure 37 shows the high level design of the physical store system.

In this high level design, the userInterface class is an interface of the classes that communicate with the user to get the SQL command and the parameters and to return the results of processing. These classes realize the Read SQL command Read User Parameter module, Read data from a text file for bulk-load module and View result module in the architecture module. The userInterface class provides to the user the error messages and allows the user also to enter parameters such as the size and the type of the buffer and the kind of constraints to apply on the database when running a bulk-load against his data. The user's constraints play major role in the performance of the system when it runs bulk-load.

The Parser class is an interface of the mySQL parser that we reused in our implementation. It realizes the DLL Parser module in the architecture. The class Parser provides all the functions to parse the DDL of SQL that the system is designed to do such as CREATE TABLE, CREATE DATABASE, INSERT, USE and LOAD INTO FILE. The result of parsing is maintained by a tree data structure, which is the parseTree class. The executionEngine class is the interface between the user interface and the other subsystems of Know-It-all project on one side and the rest of the system components on the other side. It has almost all the interfaces for all the function and the procedures of the system.

The class DataBase provides the functions to convert the information available in the parseTree to metadata, define the schema of the database, and to process the entire query against the database. Class Table provides the functions to instantiate the metadata of the table and to process the query against the table such as

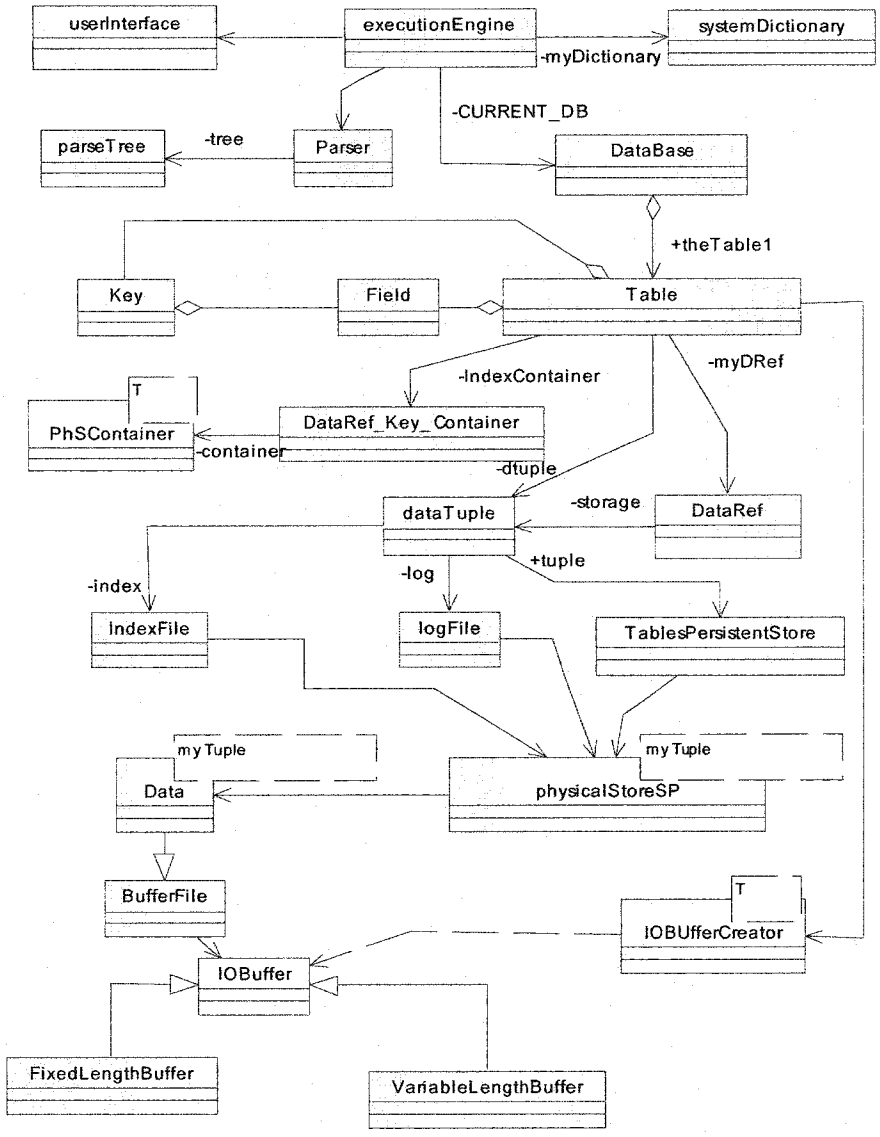


Figure 37: The High Level Design

saving and retrieving the data and bulk-loading. The classes DataBase, Table, Field and Key maintain the metadata of the database and they realize the MetaData module in the architecture module. After parsing the SQL command by the Parser and saving the information in a parseTree, the metadata is extracted from the parse tree and saved in the metadata module by the executionEngine class.

Class dataTuple provides a dynamic way to create a C++ class for the tuples of the table within the database with fields' types that meet the metadata of the table. Class dataTuple realizes the Create C++ classes dynamically module in the architecture module.

Class DataRef provides a smart pointer technology to retrieve the data from the secondary storage. It realizes the DataReference module in the architecture. The DataRef hides all the function related to saving the data in a file and retrieving the data from a file and it only retrieves the data from a file when we really need these data to decrease the I/O operation.

Class IOBuffer provides buffer management for saving data in the secondary storage and retrieving the data from the secondary storage. Buffer management classes are built using factory method design pattern where the template class IOBufferCreator plays the role of Creator class, the Buffer abstract class plays the role of Product and fixedFieldBuffer plays the role of concreteProduct in the factory method design pattern. Classes IOBuffer, fixedFieldBuffer, IOBufferCreator and variableLengthBuffer realize the Buffer Manager module in the architecture module.

Class systemDictionary is responsible for retrieving a database from the secondary store. The retrieved database is the default database and it processes the entire query against the database.

In case of the bulk-load, the DataRef_Key_Container class provides a persistent store to the data reference and the keys of a table since the memory size is not enough to hold all the data. In this container, the keys can be sorted and prepared for building indices. The index layer can iterate over this container to create the needed indexes for the table. The DataRef_Key_Container class realizes the Persistent Container module in the architecture module.

Table 4 shows the modules of the architecture and the related classes in the high level design.

Table 4: The Modules of the Architecture and the Related Classes

Module of the Architecture	Related Classes in the High Level Design
Read SQL Command Read User Parameter	userInterface
Read data from a text file for bulk load	UserInterface
View result	UserInterface
DDL Parser	Parser, parseTree
Execution Engine	ExecutionEngine, system dictionary, DataBase, Table, Key, Field, dataTuple
Create C++ Class Dynamically	dataTuple
MetaData	DataBase, Table, Key, Field
DataReference	DataRef

Table 4: Continued

Tables	TablesPersistentStore, physicalStoreSP, Data, BufferFile
Tuple	DataTuple
Log File	logFile, physicalStoreSP, Data, BufferFile
Persistent Container	PhSContainer, DataRef_key_Container, indexFile
Buffer Manager	IOBuffer, IOBufferCreator, FixedLengthBuffer, VariableLengthBuffer,

4.3. The Detailed Design

In this section, we introduce the detailed design of the major classes of the physical store. We use the UML notation to represent the class' diagrams. We document all the classes, their attributes and functionality. The global data that help to explain the behavior of the system is also documented. We explain also the main algorithm of the system. We introduce the implementation of some classes of the system and the theory concepts that we used in our implementation. We also present the scenario about the interaction of the client with the components of the system. Figure 38 depicts the details of the major classes, whereas Figure 39 shows the interface of the physical store system to the other components of KIA and to the user of the system.

In the detailed design, the userInterface class provides functions to communicate with the user such as the error messages, entering the parameters and the SQL commands. The Parser class is an interface to the mySQL parser that we reused in our implementation. The result of parsing is maintained by a tree data structure realized by the parseTree class.

The executionEngine class is the interface between the user interface and the other classes of the system. It has almost all the interfaces for all the functions and the procedures of the system.

The class DataBase provides the functions to convert the information available in the parseTree to metadata, define the schema of the database, and to process the entire query against the database.

Class Table provides the functions to instantiate the metadata of the table and to process the query against the table.

Class dataTuple provides a dynamic way to create a C++ class for the tuples of the table within the database with fields' types that meet the metadata of the table.

Class DataRef provides a smart pointer technology to hide the low level operations of saving the data and retrieving the data from the secondary storage.

Class IOBuffer provides buffer management for saving data in the secondary storage and retrieving the data from the secondary storage.

Class IOBufferCreator is a template class with factory method design pattern and provides the functions that customize the IOBuffer.

Class Key is a container of fields. It contains the metadata of the tuple's keys. This metadata is used to create the C++ key tuple class.

BufferFile is a super class that encapsulates the file operations of open, create, close, read, write, and seek in a single object.

The classes TablesPersistentStore, logFile and IndexFile are derived from BufferFile and they encapsulate the record I/O operation of the Tables, logs, and indexes in a single class.

Class systemDictionary is responsible for retrieving a database from the secondary store. The retrieved database is the default database and it processes the entire query against the database.

The DataRef_Key_Container class provides a persistent store to the data reference and the keys of a table since the memory size is not enough to hold all the data. In this container, the keys can be sorted and prepared for building indices.

Class PhSContainer is a container with an STL style. We pass to this container a persistent allocator to allocate and deallocate the elements of the container in persistent storage. In the following section, we introduce the details of the procedures, classes data members and functions in the physical store system.

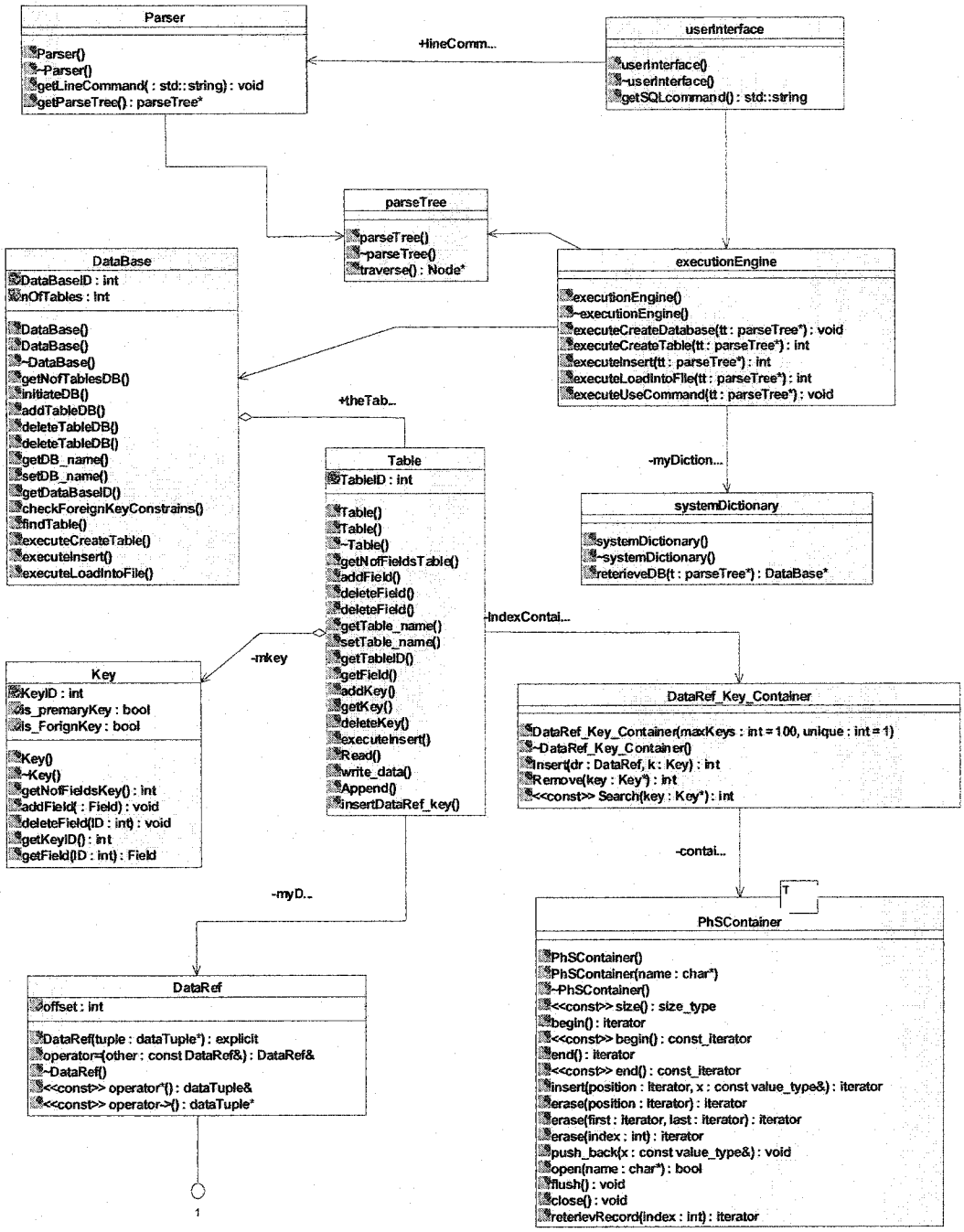


Figure 38: The Detailed Design

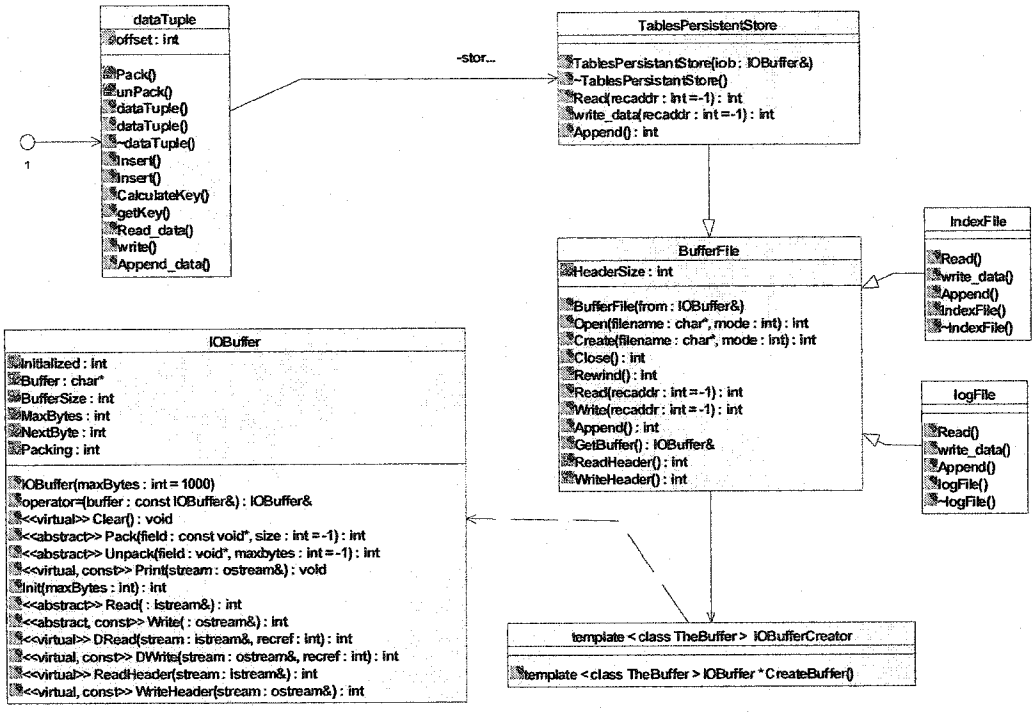


Figure 38: Continued

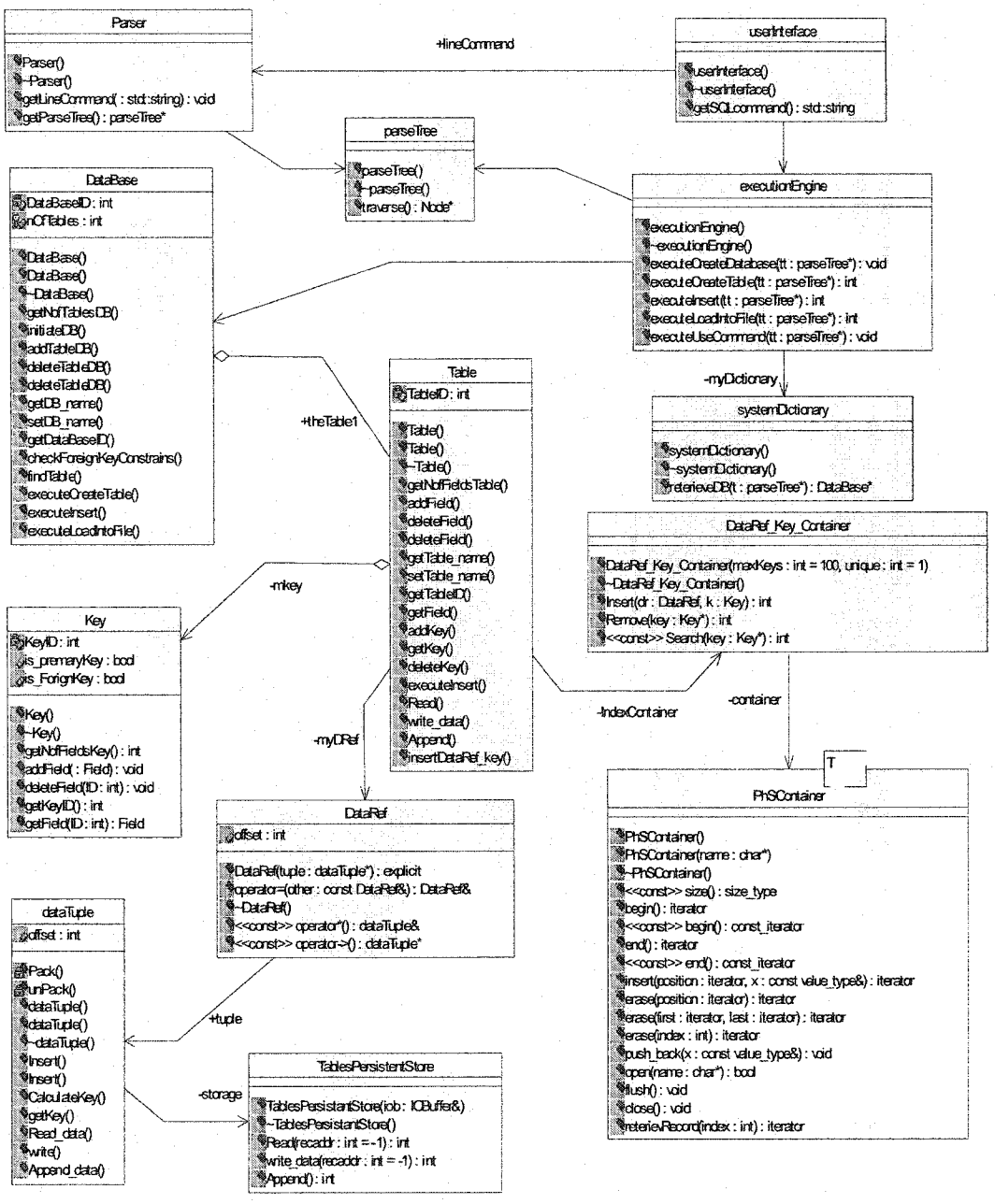


Figure 39: External Interfaces

4.3.1. Saving and Retrieving Data Using Smart Pointer Technology

The saving and retrieving data are implemented using the smart pointer technology. In the DataRef class, we overloaded the operator -> and the unary operator * to simulate the simple pointer. We can use the smart pointers to achieve a number of tasks, memory management, memory lock, and to manage the lifetime of the objects. The main purposes of the DataRef class are:

- To hide all the function behind saving the data in a file and retrieving the data from a file. The client needs only to dereference the DataRef object to retrieve the dataTuple.
- The DataRef class object only retrieves the data from a file when we really need these data by dereferencing the smart pointer DataRef. The smart pointer substitutes the dataTuple object at run time. In effect, we postpone the input/output, which are heavy weight processes of the operating system, until we really need them. This improves the use of caching techniques of the operating system since the cache memory will have more space.
- We plan to use it for locking when we add the multiple-users processing to our system.

The class IOBuffer provides a buffer of type stream of bytes. It also provides buffer management functions. Beside that the IOBuffer class provides functions to save and append the contents of a buffer in a file and in return it gives the offset of the location of the saved data in the file. It also provides functions to retrieve the data from a file using the offset of the data in the file. The offset is of type integer. The IOBuffer class uses the

system calls to the operating system to implement these functions so that the input/output function will be safe and it does not harm the operating system. Using the system calls to implement input/output function, physical store takes the advantages of the operating system's process manager, resource manager, security access, and file manager. The IOBuffer class also contains functions to pack the data from the fields into the buffer and to unpack the data from the buffer into its fields. It does the pack and unpack using two ways, depending on the type of the fields:

- Fixed field length. In this type the system knows the field length and uses this length to recognize the separation between the fields.
- Variable field length. In this type the system recognizes the separation between the fields either by a specified delimiter or by a digit in the beginning of every field shows the number of the bytes allocated to this field.

Figure 40 shows the class hierarchy of the IOBuffer class and the template IOBufferCreator. The template class IOBufferCreator is build using factory method design pattern. The factory method supports postponing the decision about the type of the buffer that fits the needs of the users and the clients. The type of the buffer can be delimFieldBuffer, lengthFieldBuffer or fixedFieldBuffer. Class fixedFieldBuffer supports fixed length records with fixed length fields. Class lengthFieldBuffer supports records where each field or record begins with a count of the number of bytes that it contains. Class delimFieldBuffer supports records that use delimiters to mark the division between records and fields. Almost all of the members and methods of these classes are identical. The only differences are in the exact packing and unpacking and in minor differences in read and write between the variable-length and fixed-length record

structures. The inheritance allows classes to share members. Virtual methods, in the base class IOBuffer, make the class hierarchy work. The class BufferFile encapsulates the file operations of open, create, close, read, write, and seek in single object. Each BufferFile object is attached to a buffer. The use of BufferFile adds a level of protection to our file operations. Once a disk file is connected to a BufferFile object, it can be manipulated only with the related buffer.

In order to save data in a file we follow the following procedure:

1- Declare an object of type IOBuffer. The type of the buffer is customized using the factory method design pattern.

2- Pack the data into the buffer as a stream of bytes. The strategy of packing the data depends on the needed type of buffer delimFieldBuffer, lengthFieldBuffer or fixedFieldBuffer.

3- Save the buffer in the corresponding file using operating system call. This system call returns the offset, which is used, latter to retrieve the data.

To retrieve the data from a file we follow the following procedure:

1- Declare an object of type IOBuffer. The type of the buffer is the same type that is used in the saving operation.

2- Read the data from the file into the buffer as a stream of bytes using the offset that we got from the saving operation.

3- Unpack the data from the buffer. The strategy of unpacking the data depends on the needed type of buffer delimFieldBuffer, lengthFieldBuffer or fixedFieldBuffer.

4- Type cast the data to its real values using C++ template typecast to insure type safety.

Figure 41 shows the sequence diagram of saving a tuple in a physical file and retrieving a tuple from a physical file. In physical store system, the tuple transfer is implemented in the class Table. First we declare an object of type IOBuffer with integer default parameter of certain capacity. The size of the buffer is customized. The type of the buffer is also customized and it can be delimFieldBuffer, lengthFieldBuffer or fixedFieldBuffer. Then we declare an object of type DataRef, which is a smart pointer that points to dataTuple object. We can use the DataRef to invoke the write function in dataTuple class. After that, the write function invokes the write function of class TablePersistentStore that invokes the Save_record function of class physicalStoreSP. The class physicalStoreSP is a proxy for class Data. After packing the data in the buffer, we use the proxy class to write the data in the file. The reading process has the same steps of writing but we have to read from the file into the buffer then we have to unpack the data from the buffer.

We studied the theory of file structure, file management, buffer management, saving and retrieving the data from a secondary store and packing and unpacking the data into and from a buffer in [FRZ98]. So, in our implementation of class IOBuffer we follow [FRZ98]. For more information, see [FRZ98].

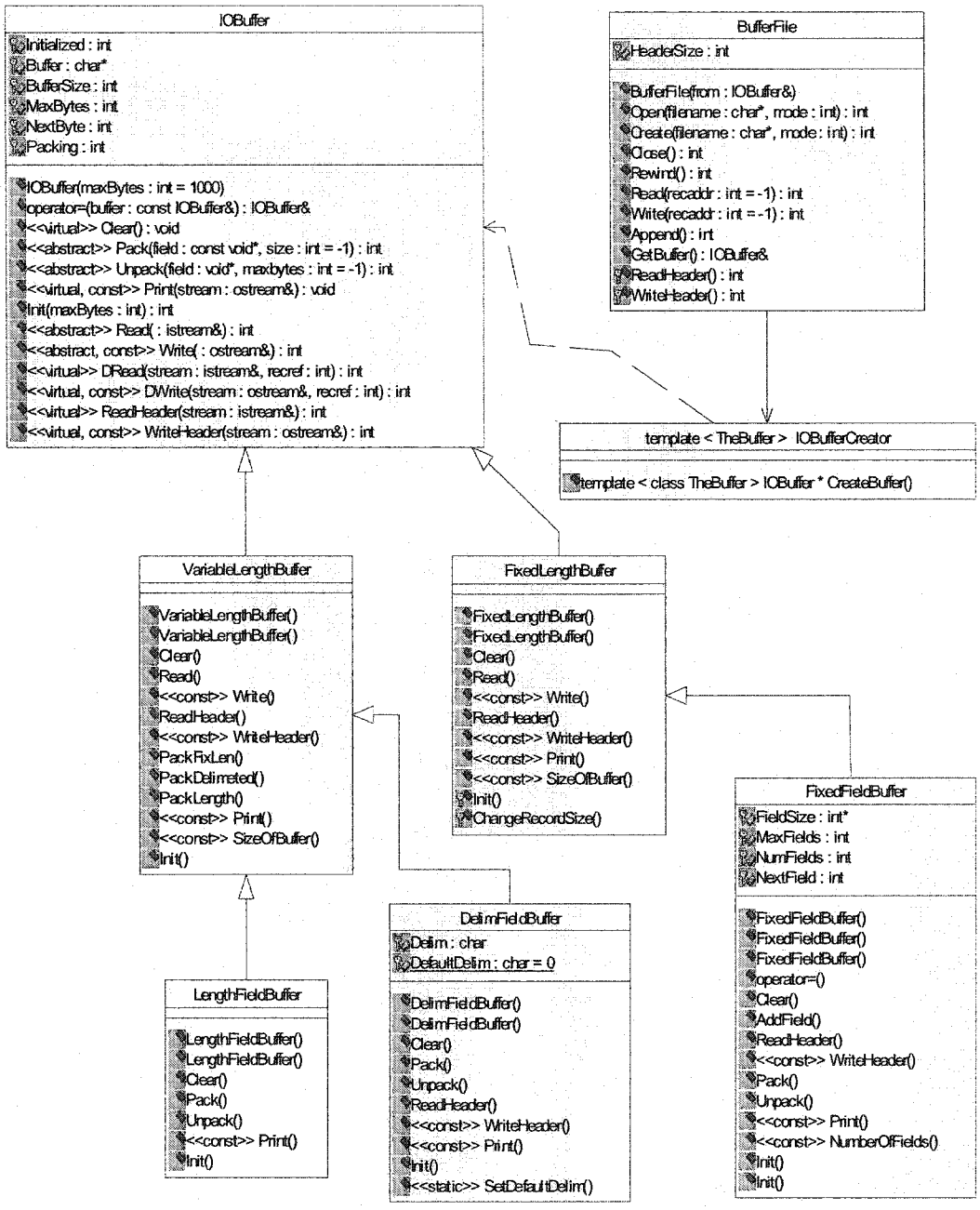


Figure 40: Buffer Classes

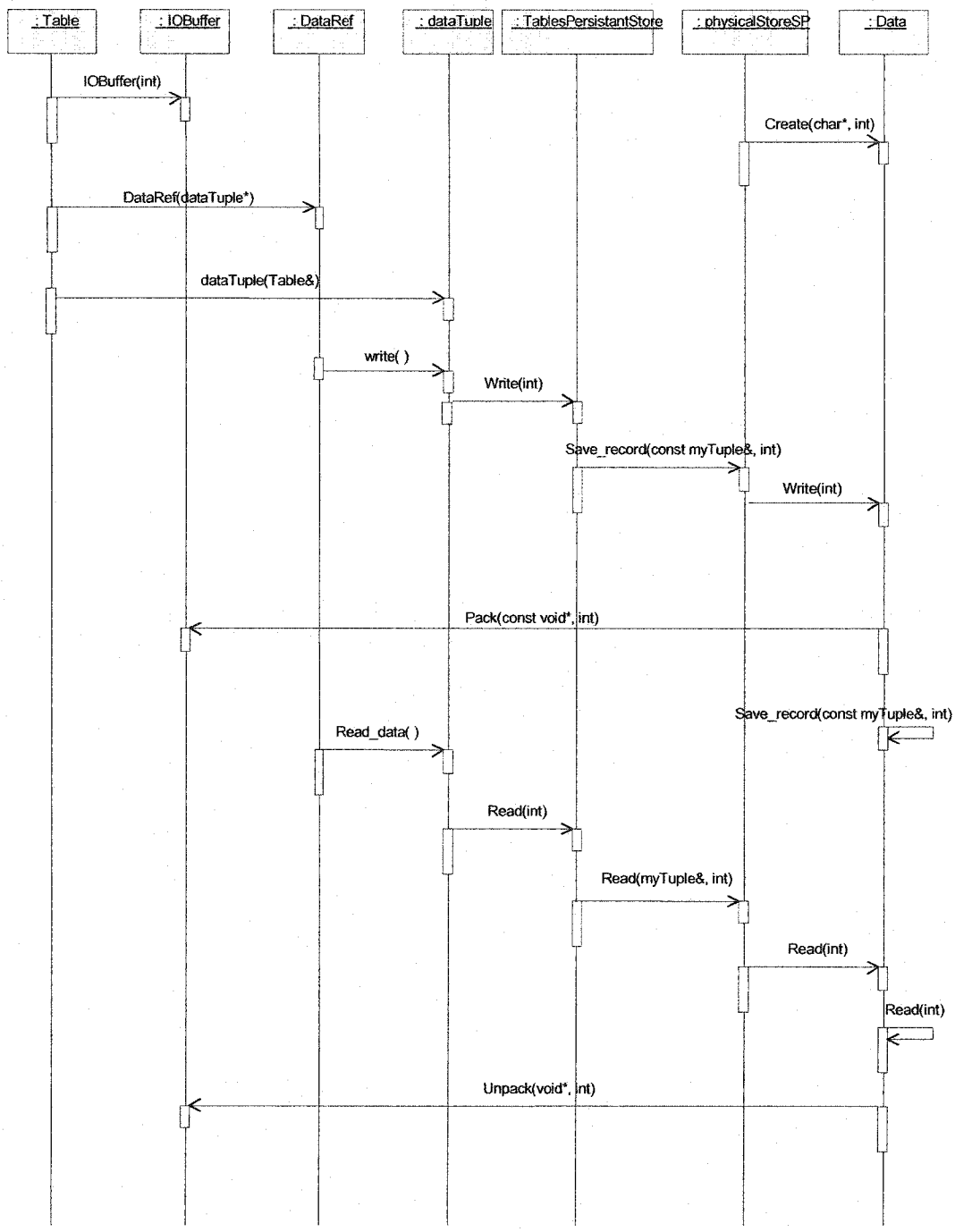


Figure 41: The Sequence Diagram for Tuple Transfer

4.3.2. The Persistent Containers

In the physical store project, we need persistent storage for the data such as the data we save in the container of the pair (DataRef, key) that the physical store is provided to the index layer to create the needed indices for the tables. The bulk-load function needs a persistent store to save temporary the pairs (DataRef, key) to large amount of sets of tuples in order to apply on these pairs external sorting by the values of their keys. Then the index layer can build the needed indexes in very efficient way. Since the pairs of (DataRef, key) are sorted, the index layer can insert them easily in their positions in the index tree. Moreover we need the persistent container to save the data of the index layer. This maintains the compatibility with the design and implementation of the index layer. We implemented an interface for a STL container that uses the persistent allocator that is available in the POST++ library. For more information, see [POST].

The main steps of designing and implementing persistent allocator are to define how to allocate and deallocate the memory of the declared object. Figure 42 shows the implementation of allocate and deallocate functions that used by POST++ library. We reuse these functions in our system. For more information, see [POST].

```
template<class _Tp> _Tp* allocate(size_t __n = 1)
{
        if (__n == 0) {   return 0;}
        storage* store = storage::find_storage((object*)this);
        if (store != NULL) {
           ArrayOfChar* a = ArrayOfChar::create(*store, __n*sizeof(_Tp));
           return (_Tp*)a->body();}
        else {    return (_Tp*)malloc(__n*sizeof(_Tp)); }
```

Figure 42: Allocate and Deallocate Functions

```
}

template<class _Tp> void deallocate(_Tp* __p, size_t __n = 1)
{
        storage* store = storage::find_storage((object*)__p);
        if (store != NULL) { store->free((object*)__p); }
          else {   free(__p);}
}
```

Figure 42: Continued

1. Allocate function: The storage is a chain of linked list of blocks. The default block size is 512 bytes. The storage uses the first fit strategy. The argument for this function is the number of the objects to allocate __n which of type size_t. In the beginning, it checks whether the number of objects equals zero; and if so, it returns zero. Then, it searches all the linked list chain of allocated blocks to determine whether the allocator itself is in the storage. If so, it means that the client has created this allocator in the storage and he wants that the objects to be stored in the persistent store. Then, the function creates a space equal to the size of the object multiplied by the number of the objects. If the allocator is not in the storage, this means that the client does not want to use the persistent store and the objects will be created on the heap only.
2. Deallocate function: The argument for this function is the pointer to the allocated space. The function searches all the allocated blocks to find the block that contains the pointer. If it is found, the storage's free function is called, otherwise the block must not have been allocated in the storage, thus the regular heap free is called instead.

4.3.3. Using Typelists Technology to Create Classes Dynamically

Actually, there are different ways to save the values of the fields of the tuple. In this section, we discuss these ways and the way is used in a DBMS (mySQL). We discuss the alternative solutions to assist our choice of saving the values of the fields of the tuple in a dataTuple class. We save the values of the fields in the file of the table where the tuple belongs and we retrieve these values from the file into their corresponding fields.

The first way to save the values of the fields is by saving every value as a stream of bytes. We get these values as strings from the parse-tree when the parser parses a user SQL command such as an INSERT command or LOAD INTO FILE command. In order to execute operations against the field value, the execution function should typecast dynamically the field value to its real type and value. The execution function gets the real type of the field by accessing the metadata object of the field.

The following algorithm explains this method:

1. Save the metadata specifying the field's types.
2. Save the values of the fields as a stream of bytes.
3. Execute operation against the field value:
   3.1 Read the type of this field from the metadata object.
   3.2 Typecast dynamically the byte stream value of this field to the real value.

The disadvantage of this way is that we do not have at run time the real value of the field. We need to type cast it to its real type every time we need to execute an operation against this value.

The second way to save the values of fields is by saving it as real object with its real types. We have two solutions to achieve that.

First, it is by generating dynamically the code of a class with attributes of the same types of the fields of the tuple and with methods to access these attributes. This class should be derived from a super class. The client can invoke the methods and access the attributes of the class through the common super class. After that, we have to compile this genertad code of the tuple and use the system call for dynamic link library (DLL) to link the generated code to the physical store system code.

We believe that this way is difficult since it needs very complicated algorithms to generate the code. Moreover it is very expensive since for every table we need to generate the code then to compile the code dynamically using DLL system calls.

Second, it is by using typelists. We studied a number of books about generic programming and pattern design and we found a very good solution to our problem by creating an object dynamically of type typelists then creating dynamically an object of type tuple with the same types of typelist. By this way we avoid the disadvantages of generating code and loading the classes dynamically mentioned above.

Creating tuple for a Table's fields

In this paragraph we explain the algorithm for Generating Tuple with Typelist that meet the fields of a table's fields. The class Table has a container of type field contains the types of all the fields of the table as a num type. The Loki library has a template function that appends a type to a Typelist or Typelist to Typelist and the type or the Typelist should be passed as a template parameters (see section 2.4.1). Consequently, We have to

build a new template append function with first template parameter with type enum and the second template parameter of type Typelist. Figure 43 shows the function append and we gave it a namespace TKH in order to use the same key words. Since the compiling of the template takes place at compile time, we have to repeat a template function for every possible type. In Figure 43, we show the template functions for three types but we have to repeat the same thing for all the possible types. The myTypes type is an enum type that contains all the possible types of a field in a table.

```
namespace TKH
{
        template <myTypes flag, typename  TList>
   struct Append
   {
   private:
     ASSERT_TYPELIST(TList);
      template<myTypes >
      struct In ;
     template<>
      struct In <string>
               { typedef TL::Append<TList,std::string > Result; };

      template<>
      struct In<integer>
      { typedef TL::Append<TList,int >Result; };

template<>
struct In<real>
      {
         typedef TL::Append<TList,double >Result;
      };
               //.....
               // continue for all other needed types
               //.....
```

Figure 43: Apend Template Function

```
/*template<>
      struct In<other>
      { typedef TL::Append<TList,other >Result; };*/
                  //.....
   public:
      typedef typename In<flag>::Result Result;
   };
}
```

Figure 43: Continued

The algorithm that creates the tuple should work as follows: In the beginning, the typlist object of the Tuple contains only the NULL type.

1. Iterate the fields' container in the Table class. For every field in the container do the following:

1.1 Check the type of the field:

1.1.1 Case it is integer Append <Tlist, int>.

1.1.2 Case it is string Append <Tlist, string>.

1.1.3 Case it is double Append <Tlist, double >.

1.1.4 .....

1.1.5 Continue until the last possible type.

2. Continue until the last field.

To realize this algorithm, since the function or the class is template, the only good way here is to build recursive template classes that call each other according to the number of the needed repetition. The function AppendTupleFunc and the classes that realize this algorithm are shown in Figure 44.

```
template<unsigned I>
class Append_N
{
   public:
     template<typename T>
     explicit Append_N (const T &x,tupleTypesContainer::iterator pos)
     {
       switch (*pos)
       {
        case integer:
        Append_N<I - 1> X(Tuple<TKH::Append<x::TList,integer> > ,pos--);
        break;
        case real:
        Append_N<I - 1> X(Tuple<TKH::Append<x::TList,real> > ,pos--);
        break;
        case string:
        Append_N<I - 1> X(Tuple<TKH::Append<x::TList,string> > ,pos--);
        break;
        //.....................
        // continue for all other needed types
        //.....................
        /*case other:
        Append_N<I - 1> X(Tuple<TKH::Append<x::TList,other> > ,pos--);
        break;
       */
       //.....................
       }
   }
};

template<>
class Append_N <0>
{
public:

template<typename T>
   explicit Append_N (const T &x,tupleTypesContainer::iterator pos)
   {
    }

};
```

Figure 44: AppendTupleFunc Function

```
template<typename T>
void AppendTupleFunc(const T &x,tupleTypesContainer::iterator pos)
{
        switch (tupleTypes.size())
        {
        case 0:
        Append_N<0 > X(const T &x,tupleTypesContainer::iterator pos);
        break;
        case 1:
        Append_N<1 > X(const T &x,tupleTypesContainer::iterator pos);
        break;
        case 2:
        Append_N<2 > X(const T &x,tupleTypesContainer::iterator pos);
        break;
        case 3:
        Append_N<3 > X(const T &x,tupleTypesContainer::iterator pos);
        break;
        // Continue the same until the maximum number of fields in a table.
                }
}
```

Figure: 44 Continued

Since Append_N class is template we have to repeat the same declaration for every possible number of fields in a table until the maximum possible number. The Tuple is passed to the constructer of class Append_N by reference. The constructor of class <0>Append_N has the resulting tuple in its constructor object that is declaired in the one step before since the <0>Append_N is the stopping condition for the recursive classes.

4.3.4. Bulk-Loading Algorithm

In this section we explain the bulk-loading algorithm of the physical store. Know-It-All is designed with scientific databases in mind, and does not provide for transactions but

instead, provides a data feed mechanism for bulk or incremental data load [BCC02]. In the physical store, we propose a technique for loading, which can be used by the index layer to create a new index, and another technique for incremental loading, which adds data to an existing table file with an index that has been created by the index layer. The main purpose is, in both techniques, to develop a solution that minimizes the amount of I/O and CPU cost since the I/O processes are heavy weight processes.

When loading a huge amount of data into database tables and its indexes, it is usually not feasible to use the standard insert operation, since this would result in a large overhead caused by unnecessary page accesses. In most cases, data are available in a text file in a random order or an order that is not suitable for the index. This random order of tuples that enter to the system will lead to random inserts where consecutive tuples will be inserted into different pages of the index. As a result, an index search and a data page access are necessary for each insertion. In the beginning of loading process there is only one data page and no random access to disk or a page split happens. However, random page accesses become frequent after subsequent page splits. Therefore, the cost for loading will not increase linearly with the number of required pages but linearly with the number of new tuples since every new tuple might to require a load of a page from the disk. In our approach, we design our bulk-loading algorithm so that the number of pages split is close to the number of new pages inserted. In KIA, we separate the index layer from the data. The separation between the data and the index layer allows us to describe two simple and efficient bulk-loading algorithms for any kind of index. The first one is for initial bulk loading and the second one for incremental bulk loading.

Description of the Algorithm

In this paragraph, we introduce a description of our version of the bulk-loading algorithm that is within the physical store. The bulk-loading algorithms can be classified according to two groups: algorithms which apply a certain partition to the multidimensional input data and load those partitions into the index; and algorithms which apply a total sort order to the input data and load the presorted data into the index [FKMBO]. In the physical store system, we use the second approach.

The bulk-loading algorithm, in the physical store, has a number of processing steps.

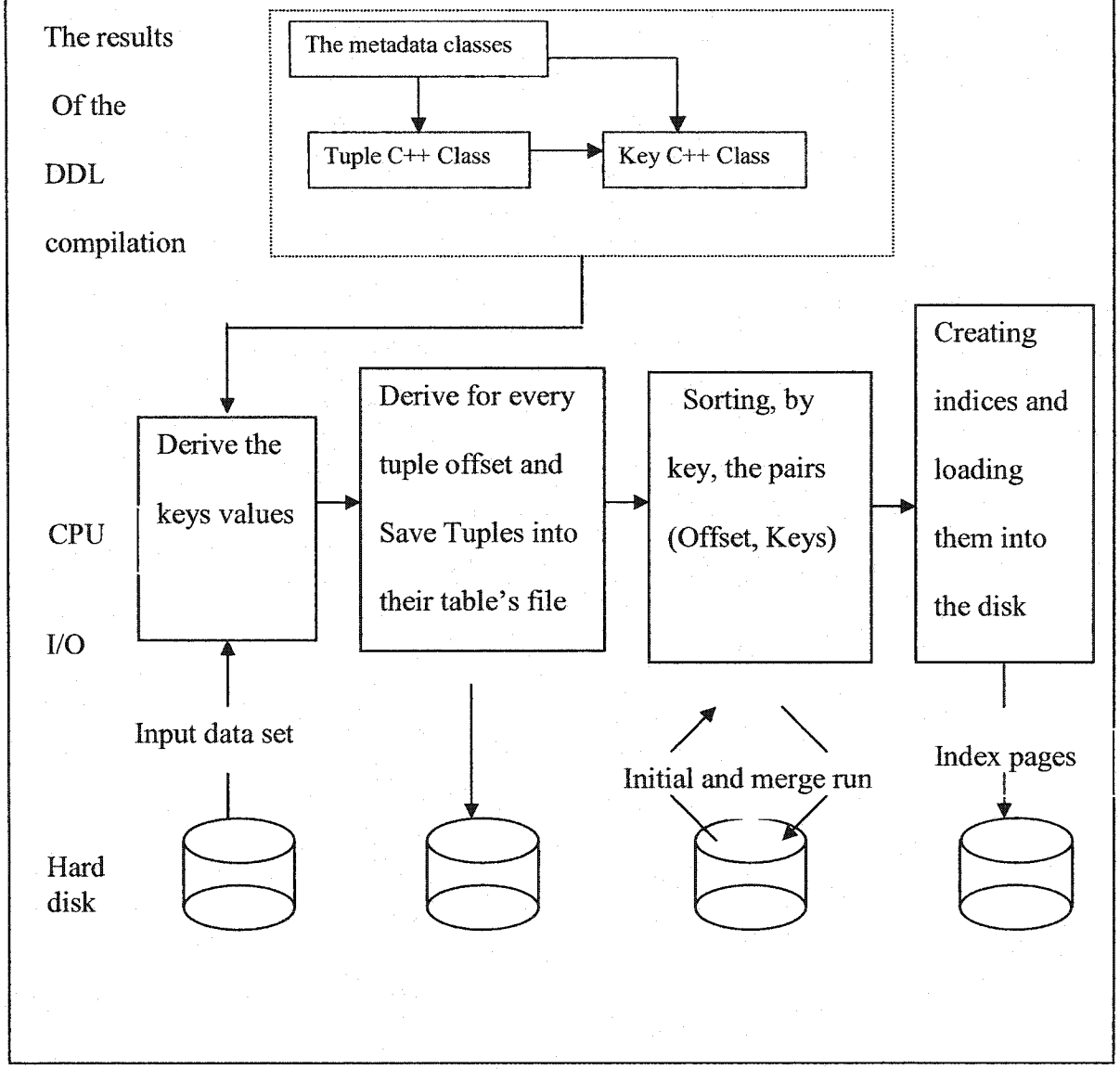
- The process starts when the user issues a LOAD DATA INFILE. The system assumes that the database is the default and the table, where the bulk-load should be loaded, has been created. If not, the system issues an error message. In effect, the metadata of the loaded data, such as the type, the number and the name of the fields, are known to the system.
- The user can specify other parameter values like the size of the buffer, the page utilization and whether it is necessary to apply all the database integrity constraints, such as the primary key, the secondary key and the foreign key. The constraints checks play a major role in the implementation time.
- In the first step, it inputs a set of tuples limited by the buffer size and the size of the tuple. Then it derives the key for each tuple of the data set.
- Secondly, for each tuple of the data set, it instantiates a DataRef object using the offset. Then it loads the data into the database and forms the pair (key, DataRef) and insert them into the container of the corresponding table.
- The pairs (key, DataRef) are sorted according to the key.

- Finally the iterator of the container is passed to the index layer to update the index. The task of the physical store finishes after forming container of the pairs of (key, DataRef).

We summarize our approach here as follows: When an I/O operation is taking place, we use the CPU to do the computations of deriving the keys and forming the pairs of (data reference, key).

In order to speed up the whole process it is useful to arrange these steps in a pipeline as depicted in Figure 45, since this avoids writing temporary results between processing steps to secondary storage, i.e., between the tuple-loading and sort steps and between the sort and the index-loading steps. Additional performance can be gained by using a binary format for the intermediate results, because it avoids conversion from internal binary representation and external ASCII representation and vice versa.

Figure 45: Bulk Loading Architecture



Deriving the Key

The key value that is used to identify a tuple is usually a subset of the attributes values of a tuple. However, internally the index might use a different representation.

In the physical store project, the key value is a C++ code that is generated dynamically using the Typelists and Tuple technology. For more information see section 4.3.3. The key class is a container of key values that holds all the key values of the tuple. After generating the C++ code for the attributes of the tuple, the key class uses the metadata to generate the C++ code for the attributes for all the keys of the tuple using the Typelists see section 4.3.3.

The External Merge Sorting

The merge sort algorithm is the best way to achieve an external sorting of large data sets [G-MUW00]. The procedure of the merge sorting is divided into two phases and works as follows:

1. The first phase:

1.1 It starts by reading, from the secondary storage, amount of data equal to the size of the RAM memory of the system. If we assume that the amount of data is N, and the size of the memory is M, then we get (N/M) number of runs.

1.2 It sorts this data, by internal sort algorithm, and writes it back into the secondary storage.

1.3 It repeats step 1.1 and 1.2 until all the data has been processed. If we assume that the size of the buffer is B, then we need $(2*(N/B))$ number of input /output operations to complete the first phase.

2. To explain the second phase, let us to assume that we use a two-way merge. In the first pass, we merge every two runs together and we sort them by internal sort algorithm. As a result, we have $(N/(M*2))$ runs.

We repeat the second phase until we have only one sorted run containing all the data. As a result, we have to repeat passes:

$\log_2 k$ times, where $k = (N/M)$

And, in every pass, we need to read and write (N/M) number of buffers. In result, we need:

$2 * (N/B)[1 + \log_2 (N/M)]$ (I/O operation)

If we have the ability to do b-way merging, then the number of I/O operations is:

$2 * (N/B)[1 + \log_b (N/M)]$ (I/O operation)

Initial Loading

Initial loading is the process of loading new data to an empty table, which does not have an index yet. We have to build the index from scratch. The initial loading algorithm works as following:

- It starts with an empty page. Now it reads tuples and adds them to this page until the page utilization reaches the specified fill-degree.
- The current page is now complete; it has the specified page utilization, and is written to disk.

- After that, for each tuple of the data set, it instantiates a DataRef object using the offset. Then it loads the data into the database and forms the pair (key, DataRef) and insert them into the container of the corresponding table;
- After that, the pairs (key, DataRef) are sorted according to the key;
- And finally the iterator of the container is passed to the index layer to create the index. The task of the physical store finishes after forming container of the pairs of (key, DataRef).
- The algorithm continues adding tuples to pages and writes the pages to disk as before, until no more tuples are left.

Incremental Loading

Incremental loading is the process of loading new data to a table that already has an index. Incremental loading differs only slightly from initial loading, because it does not only create new pages as initial loading does, but additionally it updates existing pages.

- The algorithm reads the first tuple from the input data set and checks if the key value belongs to the current index page.
- If not, it stores the current page and retrieves the needed page to which the key of the tuple belongs.
- Now it inserts the pairs of (key, DataRef) into the page and checks if its page utilization has retched the specified fill-degree.
- If so, it splits the current page in two pages. The page containing the just inserted pairs of (key, DataRef) is now the current page and the other one is written to the disk.

- The algorithm continues reading tuples and inserting the corresponding pairs of (key, DataRef) into to the current page, checking and splitting it as before until no more tuples are left. When finished, it stores the current page.

4.4. *Detail of Major Classes*

In this section we introduce the detail of the major classes that are implemented in the physical store system.

4.4.1. Global Data and Functions

Type Definition: FieldsContainer: It is an STL vector container of type Field. It is used as data member attributes in the Table class and key class.

Type Definition: KeysContainer: It is an STL vector container of type Key class. It is used as data member attributes in the Table class.

4.4.2. Class: DataBase

The DataBase class is designed to get a parseTree object and to create and instantiate the attributes of the database. The class DataBase is designed as STL container of type class Table since the database is a number of tables.

The Main Attributes and Functions of DataBase Class

Type Definition: DBtablesContainer: It is an STL vector container of type Table. We insert all the tables that belong to this database in this container.

Attributes: Tables: DBtablesContainer: It is a container for all the tables of this database.

Attributes: DatatBaseID: int: this is the database Identification number. It is unique so we can use it to identify the database in the catalogue.

Attributes: DB_name: string: The name of the database which is given by the user by the SQL command. This name is derived from the parseTree data-structure. The database name will be used as a directory name in the secondary storage that contains the files of the tables of this database.

Attributes: nOfTables: int: This is a protected attribute so that the derived Table classes can access it. Every time a new table is created this number will be increased by one and it will be decreased in case of deletion.

Attributes: Tree: parseTree: It contains all the information needed by the database to instantiate its own attributes (database name). The parser from the SQL command derives the information about the database.

Function: checkForeignKeyConstraints (): int: Invoking this function it checks the foreign keys constraints in the database. This function iterates all the tables in the containers and cheeks the foreign keys integrity.

Function: EXECUTE_CREATE_TABLE (parseTree*): int: This function is invoked by the main executer loop in order to execute a create table command within the database.

- The function algorithm:
    1. Declare an object of type Table with a parameter parseTree.
    2. Add this table object to the table container of the database.

4.4.3. Class: Table

This class is designed to hold all the meta-data information about the table and all the needed functions to process a table. The class Table points to a class DataBase since every table belongs to a definite database. It is designed to get a parseTree object, which contains the information embedded in the SQL command CREATE TABLE and it instantiates the needed attributes and fields for this Table class. It contains a container of fields that belong to this table.

The Main Attributes and Functions

Attributes: TableID: int: this is the Table Identification number. It is unique within a specific database so we can use it to identify a Table of a specific database in the catalogue.

Attributes: table_name: string: The name of the table which is given by the user in the SQL command. This name will be derived from the parseTree data-structure. The table name will be used as a name of the table file in the secondary storage under the directory with the name of the database where the table belongs.

Attribute: db: class DataBase: It is a pointer to the database where the table belongs. We need this pointer to know where we have to save the table file in the secondary storage.

Attribute: tableOfFields: Template vector container of type Field: This container is designed to insert all the metadata of fields that belong to this table. This information about the fields of the table is derived from the parseTree that contains the SQL commands issued by the user.

Type Definition: keysContainer: It is STL vector container of type fieldsContainer. It is used as attributes in the Table class.

Attribute: Keys: keysContainer: This container is design to contain all the metadata about the primary and secondary keys of the table. This information about the keys of the table is derived from the parseTree that contains the SQL commands.

Attribute: foriegnKeys: keysContainer: This container is design to contain all the metadata about the foreign keys of the table. This information about the foreign keys of the table is derived from the parseTree that contains the SQL commands issued by the user.

Attribute: Tree: parseTree: It contains all the information needed by the Table class to instantiate its own attributes (table name, fields names and properties (primary, secondary or foreign keys) and fields types. The parser from the SQL command derives the information about the database.

Attribute: mybuffer: IOBuffer: It is a buffer of bytes where we pack the tuple data in order to save it in the secondary storage and unpack the data from the buffer into the fields of tuple.

Attribute: storage: TablesPersistentStorage: This attributes provides for every table a file in the secondary storage. The file name is the same as the table name. The file will be located in a directory, which has the same name of the database where the table belongs.

Attribute: con: DataRef_Key_Container: This attributes provides for every table a container to save the data reference of each tuple with the keys of this tuple. The index layer to build the needed indexes for this table can use this container.

Function: insertDataRef_key (DataRef, keysTupleContainer): int: This function is designed to insert a pair of values of type DataRef and keysTupleContainer in the DataRef_Key_Container container. The index layer to build the needed indexes can use this container.

Function: Read (int): int: this function is designed to read a tuple from the file of the table into mybuffer object by invoking the member functions of storage object (see the members function of class TablesPersistentStore). The int parameter of this function is the offset for this tuple.

Function: write_data (int): int: this function is designed to write a tuple to the file of the table from mybuffer object by invoking the member functions of storage object (see the members function of class TablesPersistentStore). The int parameter of this function is the offset for this tuple.

Function: Append (): int: this function is designed to append a tuple to the file's end of the table from mybuffer object by invoking the member functions of storage object (see the member functions of class TablesPersistentStore).

4.4.4. Class: Field

This class is designed to hold all the meta-data information about the field and all the needed functions to process a field. The Table class derived the needed information from the parseTree to create and instantiate the needed objects of type field for every table. The information in the parseTree is derived from the SQL command CREATE TABLE which is issued by the user.

The Main Attributes and Functions

Attribute: FieldID: int: this is the field identification number. It is unique within a specific table so we can use it to identify a field of a specific table in the catalogue.

Attribute: Type: fieldType: This data member is used to keep the type of the field. There are all the needed functions to set and get this data member.

4.4.5. Class: systemDictionary

This class is designed to iterate the system catalog container to find the needed database. The findDatabase function of this class will be invoked by function findDatabase of the executionEngine class if it receives a user command USE that demanded to use a certain database as a current database instead of the default database. The function reterieveDB (), It needs the parse tree as a parameter. Parse tree usually contains all the information about the needed database.

4.4.6. Class: dataTuple

This class is designed to hold the values of the fields of the tuple.

The Main Attributes and Functions

Attribute: tb: Table*: This is a pointer to the table where the dataTuple object belongs. The dataTuple needs to access the table to know what is the type of every field in the table in order to construct the tuple data-member in its won class. The dataTuple also needs to access the file of the table to read, write and append a tuple to the table file.

Attribute: localTuple: _TUPLE: It is a structure with unnamed fields. The names of the fields is available in the fields' container in Table class. See 4.3.3, for more information about this data member.

Type definition: keysTupleContainer: It is a container of type _TUPLE. It holds the values of all the keys (primary and secondary) of the tuple. The value of every key is of type _TUPLE.

Attribute: keys: keysTupleContainer: It is a data member, which keep the keys of the tuple.

Function: Pack (): void: This function is to pack the values of the tuple in a buffer of type byte stream.

Function: unPack (): void: This function is to unpack the values from a buffer of type byte stream into its own tuple.

4.4.7. Class: myTuple

We have to mention here that the current implementation of class myTuple based on the class tuple that is available in C++ Boost [BOOST]. We studied and tested this class and all its functions in preparation for using it in KIA project. For more information about this class, see [BOOST].

We found that implementation of class tuple, in C++ Boost, does not have all the functions available in the typelist class of the Loki library. So we are thinking to substitute the class tuple of the C++ Boost library by the class Tuple of the Loki library.

4.4.8. Class: PhysicalStoreContainer

This class is implemented as an STL container with only one difference that is our container does not use the default allocator, which is provided by the STL library to

allocate memory space for the objects on the heap, instead it substitutes the default allocator with a persistent allocator that not only allocate memory space for the objects in the heap but it also allocates a space in the secondary storage to save these objects in nonvolatile memory when the container flushes the contents of its buffer. This class is designed to support:

- The bulk-load algorithm since we need a persistent store to support sorting of a large amount of data. This data consists of pairs of data reference and key for every record involved in the bulk-load.
- Creating containers for the indexes framework in the Know-it-All project. This support is compatible with the index framework design. In the index framework, the building unit of the index is the container and the container should be implemented as a persistent container [AG 01].

This class is implemented using an adapter design pattern [GHJV 97]. We converted the interface of the STL container to adapt the persistent allocator of the POST++ library. The physical store container interface is compatible with physical store system and index framework. In Figure 46, we present our implementation of the interface of the container

```
#define REDEFINE_DEFAULT_ALLOCATOR
#define USE_STD_ALLOCATORS
#define USE_MICROSOFT_STL
#include "post/post_stl.h"
#include <vector>
#include <algorithm>
#include <iterator>
#include <string>

template < class T>
class PhSContainer
{
      public:
        typedef std::vector < T > Container;
        typedef Container::iterator iterator;
        typedef Container::const_iterator const_iterator;
        typedef Container::value_type value_type;
        typedef Container::size_type size_type;
        Container* C;
        iterator ii;
        storage * sto4;

        PhSContainer ()
            {
            }
        PhSContainer (char* name )
        {
                sto4=new storage(name);
                 if (sto4->open(storage::fixed))
                                C=(Container*)sto4->get_root_object();
                else { cerr << "Failed to open storage2\n"; }
if (C == NULL)
                        {
                                C = new (*sto4) Container;
                                sto4->set_root_object((object*)C);
                        }
        }
```

Figure 46: Physical Store Container Class

```
        ~PhSContainer()
          {
                                delete C;delete sto4;
          }
         size_type size ( ) const
            {
               return C->size();
            }
        iterator begin ( ) {return C->begin ( ) ;}
        const_iterator begin ( ) const {return C->begin ( ) ;}
        iterator end ( ) {return C->end ( ) ; }
        const_iterator end ( ) const{return C->end ( ) ;}
        iterator insert (iterator position, const value_type& x) {return c.insert(
position,  x);}
        iterator erase (iterator position) {return C->erase ( position);}
        iterator erase (iterator first, iterator last) {return C->erase ( first,last);}
        iterator erase (int index) {
             if ((C != NULL)&& ( (C->begin()+index)<(C->end()))) { return C-
>erase ( C->begin()+index);} else return 0;}
        void push_back(const value_type& x){C->push_back(*new (*sto4)
value_type(x) );}
        bool open(char* name){ sto4=storage("name");}
        void flush(){ sto4->flush();}
        void close(){sto4->close(); }
        iterator reterievRecord(int index){
            if ((C != NULL)&& ( (C->begin()+index)<(C->end())))  { return ii=
C->begin()+index;} else return 0;}
};
```

Figure 46: Continued

4.4.9. Class: DataRef

This class is implemented using the smart pointer technology. In the smart pointer class, we overloaded the operator -> and the unary operator * to simulate the simple pointer. We can use the smart pointers to achieve a number of tasks, memory management, memory lock, and to manage the lifetime of the pointed to objects.

In Figure 47, we present our implementation of the DataRef class. This implementation of a smart pointer satisfies the recent stage of physical store prototype. In the next stages, we can add the other capabilities of smart pointer such as template smart pointer, ownership policy and multithreading policy. Adding these capabilities is straightforward and it is available in a lot of references. Please see section 2.5.3.

```
class DataRef
{
        private:
                dataTuple * tuple;
         public:
                explicit DataRef (dataTuple * tuple): tuple (tuple) {}
                ~DataRef(){}
                DataRef & operator =(const DataRef& other){}
                dataTuple & operator * () const{
                 return *tuple;}
               dataTuple * operator -> () const{
               return tuple;}
};
```

Figure 47: DataRef Class

5. Conclusion and Future Work

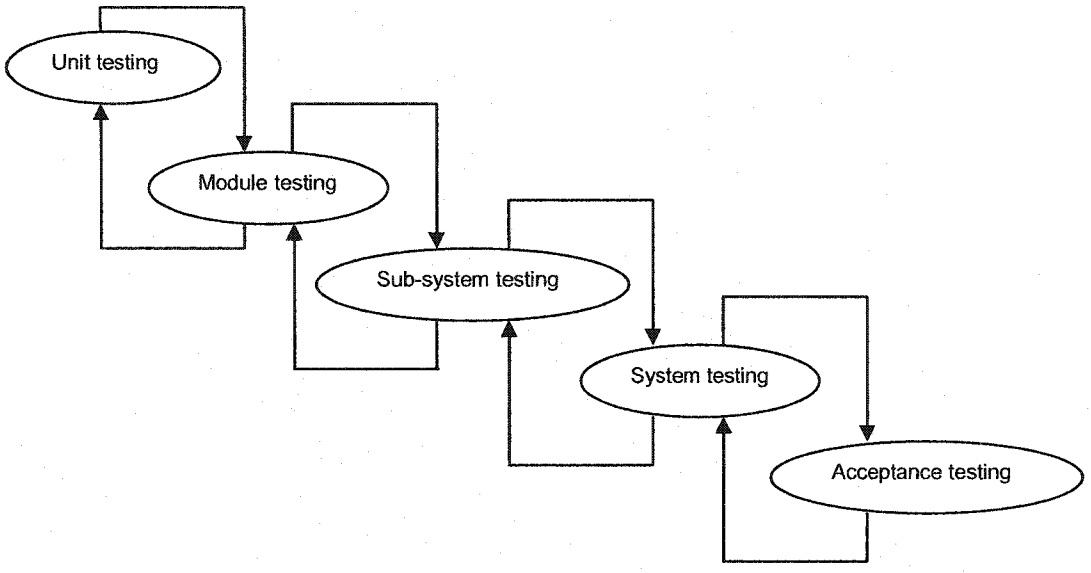
In this chapter, we present the conclusion of our work. The project is incomplete. The mySQL parser is not integrated in the system yet, so we used placeholder classes Parser and parseTree to verify the system. The Creating class dynamically module is not integrated in the system yet and we used a Boost tuple as a placeholder to test the implemented part of the system. This chapter also presents the details of the status of the work to show the parts that have been completed and the parts not yet completed. We present the testing of the implemented modules and classes in the physical store system. Then, we present the contribution of our work and the future work.

5.1. Testing

We introduce in this section testing for the procedures and tasks that we have implemented within the system.

Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system. Figure 48 shows a five-stage testing process [SOI 01]. In this testing process individual components are tested, the modules, which are collecttions of dependent components, are tested, sub-systems, which are a collection of modules, are tested; system testing involves the integration of the sub-system and acceptance testinmg involves testing with the customer data rather than simulated test data. The testing process is an iterative with information being fed back from later stages to earlier parts of the process.

Figure 48: The Testing Process [SOI 01].



In this stage of implementing the physical store system, we can apply only the unit, module, and sub-system testing.

Testing The Persistent Storage Module

In order to test this module we build a class Recording that contains attributes to describe some selected information about a company that produces music CDs. We create about 50 objects of the Recording class to test the Persistent Storage Module. We save these records in a specified file and we save the offset of each record that shows its location in the file. We use the offset to retrieve the records from the file. Then we do a search test that simulates the index layer and we retrieve the needed record from the file and indicate the location of the file.

In our implementation of the physical store system, we provide for the clients a number of ways of creating persistent storage objects for database.

- Creating a persistent storage using class PhysicalStoreSP:

Class "physicalStoreSP.h" is implemented as a Proxy class. This Proxy class postpones the heavy weight processes of I/O operations until we actually need to use it.

1- Declare an object of type IOBuffer: For example create DelimFieldBuffer Buffer;

2- Declare an object of type PhysicalStoreSP:

```
PhysicalStoreSP<Recording>  Rec2file (Buffer);
```

3-Pass the name of the Data-File to the PhysicalStoreSP object

```
Rec2file . Create ("record2.dat", ios::out);
```

4- Declare an object or a number of objects (according to the capacity of a page of records) of the type of the client's tuple of records.

```
Recording * R[10];
```

R[0] = new Recording ("LON", "2312", "Romeo and Juliet", "Prokofiev", "Maazel");

5- Save this Records in the physical file:

```
Data_ref = Rec2file . Save_record (*R[0]);
```

Now the procedure is ready to save and retrieve data from the file.

- Example of output: In this output the procedure prints the records already in the file. Then it derives for every record its key and prints every key with its location in the file. Then it searches for a record using its key, finds it and prints it with its location in the file. Figure 49 shows this output.

```
17    LON|2312|Romeo and Juliet|Prokofiev|Maazel
62    RCA|2626|Quartet in C Sharp Minor|Beethoven|Julliard
117   WAR|23699|Touchstone|Corea|Corea
152   ANG|3795|Symphony No. 9|Beethoven|Giulini
196   COL|38358|Nebraska|Springsteen|Springsteen
241   DG|18807|Symphony No. 9|Beethoven|Karajan
285   MER|75016|Coq d'or Suite|Rimsky-Korsakov|Leinsdorf
338   COL|31809|Symphony No. 9|Dvorak|Bernstein
382   DG|139201|Violin Concerto|Beethoven|Ferras
427   FF|245|Good News|Sweet Honey in the Rock|Sweet Honey in the Rock
Text Index max keys 10 num keys 10
      Key[0] ANG3795 RecAddr 152
      Key[1] COL31809 RecAddr 338
      Key[2] COL38358 RecAddr 196
      Key[3] DG139201 RecAddr 382
      Key[4] DG18807 RecAddr 241
      Key[5] FF245 RecAddr 427 Key[6] LON2312 RecAddr 17
      Key[7] MER75016 RecAddr 285
      Key[8] RCA2626 RecAddr 62
```

Figure 49: An Output Using physicalStoreSP Class

```
    Key[9] WAR23699 RecAddr 117
Retrieve LON2312 at recaddr 17
read result: 17
Found record: LON|2312|Romeo and Juliet|Prokofiev|Maazel
```

Figure 49: Continued

Another way of creating a persistent store object is to use class "physicalStoreContainer.h". This Class was implemented using an STL container and using a persistent allocator instead of the default allocator. In this class we use the facilities of the POST persistent object storage. The advantage of this way is that we can use all STL facilities and the algorithms of the STL class <algorithm> to process the elements of the container efficiently. Using this class is similar to using any STL container.

In order to test this module we create a class Personnel that contains information about ID, names and address. We create about 50 objects of class Personnel and save this information in a text file in order to simulate the bulk loading. We tell our system read this information from the text file and save it in a persistent container. Then we tell our system show us the saved records. We delete some records and ask the system to show the remaining records. We also ask the system to retrieve a specific record with a specified index.

- Example of output: The procedure starts by printing out all the records that are saved in the file. After that the procedure delets one record and prints the

remaining records in the file. Then it asks to input more records. Figure 50 shows an example of the output.

```
The objects already in the file are:
(1111 John Smith 122 Bordeaux)
(333333 Gregory Attwood 154 St Laurent)
(223333 Andrea Attwood 454 St Laurent)
(113333 Gregory Aggasi 154 St Catharine)
(333654 Gregory Jones 4354 St Laurent)
(333098 Albert Attwood 544 St Denis)
(333098 Albert Attwood 544 St Denis)
In the file, after removing the second element
(1111 John Smith 122 Bordeaux)
 (223333 Andrea Attwood 454 St Laurent)
(113333 Gregory Aggasi 154 St Catharine)
(333654 Gregory Jones 4354 St Laurent)
(333098 Albert Attwood 544 St Denis)
(333098 Albert Attwood 544 St Denis)
Add some tuples. Terminate input with non-tuple format2.
```

Figure 50: An Output Using physicalStoreContainer Class

```
(444444 John Linon  768 Papineau)
Input the index of the tuple to be reterieved:
1
the tuple is found: (1111 John Smith 122 Bordeaux)
```

Figure 50: Continued

5.2. *The Status of the Work*

In this section we introduce the status of the physical store project. We do not claim that we did a complete implementation for the system. We plan to continue our work in our future direction.

5.2.1. The Completed Parts

The specification of the system is complete. It includes the specification of the interface and the data definition language, the main tasks and functions provided for the users and for the other layers and subframeworks of the Know-It-All framework. The physical store supports the separation between the index framework and the physical store system. The index framework and the physical store system communicate through the data reference and the keys.

The design is complete all the functions have been designed and placeholder classes and methods have been put with return types to validate the system design. The design includes the architecture of the physical store and detailed object-oriented design for the

physical store system. We discussed the structural aspects of this project, the overall structure, and the detailed design.

Creating classes dynamically is designed and implemented; it is in the testing stage. We implemented the algorithm that creates an object dynamically of type typelist then creating dynamically an object of type tuple with the same types as are in the typelist.

Bulk-loading algorithm is designed and implemented as an STL persistent container. This meets the index layer requirements. The bulk-load algorithm needs a persistent store to save temporarily the pairs (DataRef, key) of large sets of tuples in order to perform an external sort by the values of their keys

Saving and retrieving the index data is implemented as a persistent container. We tested this implementation to save and retrieve records in a persistent container.

Saving and retrieving a tuple into a physical file is implemented and tested completely.

The template factory method is implemented and tested to customize the buffer. In this implementation the buffer can be customized according to the application.

A proxy design pattern is implemented and tested for saving and retrieving data from a file.

A smart pointer is implemented and tested. The smart pointer is designed to point to the tuple. This meets the index layer requirements.

Dynamic linking using a class factory is implemented and integrated in the system. We plan to use it for dynamically linking the user's type (This is not in the requirements at this stage). We also can use it to dynamically link the user's C++ tuple classes.

The class catalogue to manage the catalogue table that holds the attributes of the catalogue is implemented. The tuple of this table does not need to be created dynamically since all its data members are known.

Class logFile to manage the log table is implemented. The tuple of this table does not need to be created dynamically since all its data members are known.

The software process that we follow:

- Specification: completed.
- Design: completed.
- Prototype with placeholders: completed and tested.
- Prototype. A number of its parts have been completed.
- Complete Implementation: Some parts have been completed.

5.2.2. The Limitations

In this section, we present the work still to be done in the physical store. We also present a time table shows the estimated time needed to complete each part of the work to be done in the project.

The mySQL parser is not integrated with the system yet. In the design, the classes Parser and parseTree are implemented as interfaces to the mySQL parser. To verify the design of the system, we put placeholder functions with return types in these classes.

Creating classes dynamically is not integrated with the system yet. We use a Boost tuple as a placeholder to test the implemented part of the system.

The user Interface is not implemented completely.

System recovery is not implemented since it is not in the specification at this stage.

The catalogue manager is not implemented since it is not in the specification at this stage. Strategies to optimize external sorting are not implemented.

In my opinion, the complete implementation of the project is beyond a master's thesis. Table 5 shows our estimation of the time needed to complete each part of the physical store system.

Table 5: The Estimated Time

The Work to be Done	**The Estimated Time (For one person)**
Designing the catalogue manager	3 months
Implementing and testing the catalogue manager	4 months
Documenting the catalogue manager	2 months
Integrating the mySQL parser	2 months
Testing the integrated mySQL parser	1 month
Integrating Creating classes dynamically	2 months
Testing Creating classes dynamically.	2 months
Designing the strategies to optimize external sorting.	1 month
Implementing and testing the strategies to optimize the external sorting	2 months
Documenting the strategies to optimize external sorting	1 month
Designing the complete system test plan	2 months
Implementing the testing plan	2 months
The total estimated time	**24 months**

5.2.3. The Application Programming Interface Status

In this section, we introduce the status of the Application Programming interface:

1. API: Create database

The function to create a directory with the name of the database is implemented. The class to save its metadata is implemented. The function to initialize the created database as the current database is implemented. The mySQL parser is not integrated into the system yet.

2. API: Create table

The function to create a file with the name of the table is implemented. The class to save its metadata is implemented. The function to create tuples dynamically is implemented but not tested. The mySQL parser is not integrated into the system yet

3. API: USE

The function that saves the metadata of the current database in a catalogue file is implemented. The function that initializes the USE database as the current database, is implemented. The function to retrieve the USE database from the catalogue File is implemented (with no index). The mySQL parser is not integrated into the system yet.

4. API: Enter parameters

The function to read a number of parameters from the user and to save them in order to use them in the processing is implemented.

5. API: load data into file

The STL persistent container is implemented. Customized external sorting is not implemented. The mySQL parser is not integrated into the system yet.

6. API: Create index

There are three Masters student, in our team, preparing theses on index. The mySQL parser is not integrated into the system yet.

7. API: Dereference a DataRef

The complete procedure is implemented.

8. API: Derive the key

The function to create the key tuple dynamically is implemented but not tested. The function to unpack the data from the tuple to the key is implemented.

9. API: Insert

The function to save the data in the created tuple is implemented. The function to save the tuple is implemented.

10. API: Initial load

The STL persistent container is implemented. Customized external sorting is not implemented.

11. API: Incremental load

The STL persistent container is implemented. Customized external sorting is not implemented.

5.3. Contributions

This thesis presents an illustration of how to use typelists, techniques embedded in STL, and design patterns to design and implement a high quality physical store system. In our physical store system, we implemented and used several design patterns such as proxy, smart pointer, iterator and Factory method design pattern. We used these patterns to solve the problems we faced and to make the system more reusable, extensible and flexible. We

introduced the detailed design, the prototype, the definition of the specification of the physical store, the implementation of the persistent store, and description of the design of the bulk load in the KIA project. A number of procedures to test the implemented classes are also presented.

5.4. Future Direction

In our future works, we are planning to convert the prototype to a complete high quality physical store system. We also want to extend the system to be customizable by the user to process new types of data designed by the user. Dynamic linking libraries are separate files containing classes that can be called by programs and other DLLs to perform certain tasks such as processing new types of data designed by user. Dynamic linking provides a mechanism to link physical store system to libraries at run time. The user provides his new type classes that should be compiled and reside in their own executable files. The operating system loads the DLL into memory when the physical store system calls the dynamic liking mechanism. The classes of user's types should be written by specific rules that put another restrictions on the way the type classes should be written. The specification of the user's type classes is also one of our future works. Beside that we think that it is important to expand the system for multi-users by adding the transaction manager to make sure the every transaction in its queue will be allocated the needed resources and it will be executed properly. The concurrency manager takes care of multiprocessing issue and that all the resources are functioning efficiently. In order toachieve concurrency the concurrency manager uses the locks to lock the tables. We also have to add to the system the ability to process the distributed database by involving communication protocols.

Bibliography

[A 02] Andrei Alexandrescu, Modern C++ Design.Addison-Wesley, 2001.

[AG 01] Ashraf Gaffar, Design A Framework For Database Indexes, Master thesis, Concordia University, 2001,

[BB 02] Michael Boggs, Wendy Boggs, Mastering UML with Rational Rose 2002, Sybex, Incorporation, 2002.

[BCC 02] Greg Butler, Ling Chen, Xuede Chen, Ashraf Gaffar, Jinmiao Li, Lugang Xu, The Know- It- All Project: A Case Study in Framework Development and Evolution, Domain Oriented Systems Development: Perspectives and Practices, Kiyoshi Itoh, Satoshi Kumagai, T. Hirota (eds), Taylor and Francis Publishers, UK, 2002, pp. 101-117.

[BOOST] URL: http://www.boost.org

[CWFH 94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L.McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan,Odysseas G. Tsatalos, Seth J. White, Michael J. Zwilling: Shoring Up Persistent Applications. SIGMOD Conference 1994: 383-394

[FKMBO] Robert Fenk, Akihiko Kawakami, Volker Markl, Rudolf Bayer, Shuichi Osaki. Bulk loading a data warehouse built upon a UB-Tree. International Database Engineering and Application Symposium (IDEAS'00), 2000. IEEE, 2000, p. 179.

[FRZ 98] Michael J. Folk, Greg Riccardi, Bill Zoellick, File Structure: An Object-Oriented Approach with C++. Addison-Wesley, 1998.

[GHJV 97] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns Element of Reusable Object-Oriented Software. Addison-Wesley, 1997.

[G-MUW00] Hector Garcia-Molina, Jeffry D. Ullman, Jennifer Widom, Database System Implementation. Prentice Hall, 2000.

[GS93] David Garlan, Mary Shaw, An introduction to software architecture, In Advanced in Software Engineering and Knowledge Engineering, volume 1. World Scientific Publishing Company, 1993.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. Generalized Search Trees for Database Systems. Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, September 1995, pp. 526-573.

[J 99] Nicolai M. Josuttis, The C++ Standard Library A Tutorial and Reference, Addison-Wesley, 1999.

[JGJ 97] Ivar Jacobson, Martin Griss, Patrik Jonsson. Software Reuse Architecture, Process and Organization for Business Success. ACM Press, 1997.

[JJ 02] J. Jarvi. Proposal for adding tuple types into the standard library: Programming language C++. Technical Report N1403=02-0061, ISO/IECJTC1, Information technology, Subcommittee SC 22, 2002.

[L 97] Kenneth C. Louden, Compiler Construction Principles and Practice. PWS Publishing Company, 1997.

[MAU99] Matthew H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison –Wesley,1999.

[POST] URL: http://www.garret.ru/~knizhnik/post/readme.htm

[R 99] Steven P. Reiss, A Practical Introduction to Software Design with C++, John Wiley & Sons, 1999.

[RG 00] Raghu Ramakrishnan, Johannes Gherke. Database Management System, McGraw-Hill, 2000.

[SGG00] Abraham Silberschatz, Peter Galvin, Greg Gagne, Applied Operating System Concepts, John Wiley & Sons, Inc, 2000.

[SH 97] Clifford A Shaffer, A Practical Introduction to Data Structure And Algorithm Analysis. Prentice-Hall, Inc, 1997.

[SHA 99] William A. Shay, Understanding Data Communications & Networks, ITP, 1999.

[SOI 01] Ian Sommerville, Software Engineering, Addison-Wesley 2001.

[RO00] Robert Robson. Using the STL: the C++ Standard Template Library, second edition, Springer-Verlag, 2000.

[WS98] George Fluger, William A. Stubblefield, Artificial Intelligence, Addison-Wesley, 1998.

[WA02] Michel "Monty" Widenus, David Axmark, and MySQL AB. MySQL Reference Manual Documentation from the Source. O'Reilly Community Press, 2002.

Abbreviations

ADT: Abstract Data Structure.

API: Application Programming Interface

COW: Copy On Write strategy

CPU: Central Processing Unit.

DBA: Database administrator.

DBM: Database Management System.

DDL: Data Definition Language.

DLL: Dynamic Loading Library.

FAT: File Allocation Table.

FIFO: First-in-First-out policy.

GiST: A Generalized Search Tree for Secondary Storage: It is a balanced tree structure for searching a secondary storage.

IC: Integrity Constraints.

IDL: Interface Definition Language.

I/O: Input/Output.

KIA: Know-It –All Project is a framework for database management system that supports a variety of data models of data and knowledge, the integration of different paradigms and heterogeneous database.

LALR (1): Lookahead LR (1) parsing.

LIFO: Last-in-first-out policy.

ODL: Object Data Language.

ODMG: Object Database Management Group.

OML: Object Manipulation Language.

OODBMS: Object-Oriented Database Management System.

OQL: Object Query Language.

ORDBMS: Object Relational Database management system.

SDL: SHORE definition language.

SQL: Structured Query Language.

STL: Standard Template Library: It is the most modern version of C++.

URL: Uniform Resource Locator.

UML: Unified Modeling Language.

YACC: Yet Another Compiler Compiler.