Component-Based Generalized Database Index Model

Ashraf Gaffar
Concordia University, Canada

INTRODUCTION

The performance of a database is greatly affected by the performance of its indexes. An industrial quality database typically has several indexes associated with it. Therefore, the design of a good quality index is essential to the success of any nontrivial database. Parallel to their significance, indexed data structures are inherently complex applications that require a lot of effort and consume a considerable amount of resources. Index frameworks rely on code reuse to reasonably reduce the costs associated with them (Lynch & Stonebraker, 1988; Stonebraker, 1986). Generalized database systems have further addressed this challenge by offering databases with indexes that can be adjusted to different data/key types, different queries, or both. The generalized search tree (GiST; Hellerstein, Naughton, & Pfeffer, 1995; Hellerstein, Papadimitriou, & Koutsoupias, 1997) is a good example of a database system with a generalized index, or generalized index database for simplicity. Additional improvements extended the concept of generalized index databases to work on different domains by having generalized access methods (search criteria). For example, based on the work of Hellerstein et al. (1995), Aoki (1998) provides a generalized framework that allows users to adjust the index to different search criteria like equality, similarity, or nearest neighbor search. This makes the database system customizable to not only finding the exact records, but also to finding records that are “similar” or “close” to a given record. Users can customize their own criteria of “similarity” and let the index apply it to the database and return all “similar” results. This is particularly important for more challenging domains like multimedia applications, where there is always the need to find a “close image,” a “similar sound,” or “matching fingerprints.”

The common drawback of all these improvements is having a monolithic code that allows users to only adjust the few lines of code that are meant to be customized. Outside these areas, code is often bulky and difficult to maintain or upgrade. However, with the increasing dependence on software engineering techniques, these problems can be ratified. Component-based frameworks provide solid ground for future maintenance, replacement, and additions to the code in an orderly fashion that does not reduce the quality of the system (the aging effect). Customization does not have to be limited to few lines of the code. In our work, we provide a new model to redesign the generalization of database indexes using components. Unlike previous works, the customization is not done at the code level, but rather at higher conceptual levels from the early design stages. Our design is based on total decoupling of the design modules and connecting them through well-defined interfaces to build the database system. All this is done at the design level. After the design is completed, implementation is done by obtaining the already-existing commercial off-the-shelf (COTS) components and connecting them together according to the design model. Following this design-level customization paradigm, the system can be customized as usual by the user at prespecified locations (predefined few lines of code), but more importantly, a large-scale customization is also possible at the design level. The system designer can redesign the generalized model by adding, removing, or replacing a few components in the design model to instantiate the model into a new concrete design. Afterwards, only the affected components of the source code need to be replaced by new ones. In our system, COTS components are extensively used, which dramatically reduces the cost of development. We needed to implement a few components with the same predefined interfaces where COTS components were not suitable, for example, when special concurrency control management or specific storage needs were necessary. This adds more flexibility to the model.

BACKGROUND

Component-based software systems have received a lot of attention recently in several application domains. Reusing existing components serves the important goals of simultaneously reducing development time and cost (time-to-market) and producing quality code. Moreover, components can be added, removed, or replaced during system maintenance or reengineering phases, which leads to faster and better results (Sanlaville,
Faver, & Ledra, 2001). Quartel, Sinderen, and Ferreira Pires (1999) offer some useful techniques to model and analyze software components and their composition. To promote successful reuse, components have to be context-independent (Schmidt, 1997). C++ has adopted a component-based library as its standard library (STL, the Standard Template Library), which provides programming-level components that can be used at the design stage and later added as code fragments to generate a considerable part of the application code (Musser, Derge, & Saini, 2001). STL adopts a new paradigm that separates software applications into a few component types for storage and processing. Using the STL paradigm at the early phases of conceptual design of the generalized database indexes facilitates the design and implementation of the whole systems by iteratively adding arbitrary combinations of STL types.

**STL BUILDING BLOCKS**

We built a modular framework for a generalized database by applying the STL (ANSI C++) modularity concept (Austern, 1999) in the analysis, architecture, design, and interface stages. We developed a new set of models for an index in different application domains; linearly ordered domain, general domain (with both depth-first and breadth-first access methods), and eventually similarity search domain.

Using coherent, decoupled building blocks allows us to locate and replace a few blocks in each model to obtain a new model with no major impact on the rest of the system design. This makes modifications limited to specific regions of the system. A wealth of STL modules can be used in the replacement process. The framework also allows for introducing new compatible modules as needed.

STL is based on separating **algorithms** from **data structures** as two different types of generic building blocks (Breymann, 2000). It also introduces other generic building blocks to complete all the abstract design blocks (e.g., iterators, functors, container adaptors). They work as “adaptors” or “glue” to allow for building a large system using arbitrary combinations of these blocks. This emphasizes code reuse, a fundamental concept in modern software engineering. STL allows for new building blocks to be written and seamlessly integrated with the existing ones, thus emphasizing flexibility and extendibility. This makes STL particularly adaptive to different programming contexts, including algorithms, data structures, and data types. We briefly explain some of the major building blocks of STL that we used in the design and implementation of the index system.

**Containers**

A container is a common data structure that stores a group of similar objects, each of which can be a primitive data type, a class object, or—as in the database domain—a data record. The container manages its own objects: their storage (see the Allocators section) and access (see the Iterators section). The stored objects belong to the container and are accessed through its interface. Each container provides a set of public data members and member functions to provide information and facilitate the access to its elements. Different container types have different ways of managing their objects. In other words, they offer different sets of functionality to deal with their objects.

**Iterators**

Containers do not allow direct access to their stored objects, but rather through another class of objects called iterators. An iterator is an object that can reference an element in a container. It allows programmers to access all elements in a sequential, random, or indexed way depending on the type of the container.

**Algorithms**

Iterators separate the processing logic from the container types and element types stored inside them. The same algorithm, written once, can be applied to different containers with different stored elements by having the algorithm code deal with iterators that can access the containers. This allows for a complete implementation of generic algorithms, like search and sort, without knowing the exact type of the container (its functionality) or the type of elements stored in it.

**Allocators**

As we know, containers are responsible for managing both the access and the storage of their elements. At the front end, the access is allowed by providing “suitable” iterators and a set of functions to provide information about the elements, to return a reference to an element, or to add/delete an element. At the back end, the physical storage remains an essential part to the container, allocating memory to new elements and freeing memory from deleted elements. Normally the user need not worry about storage management when using a container. This is done “behind the scenes” without any user intervention. This greatly simplifies the use of containers. Frequently, however, some applications require a