1995

An Object-Oriented Heterogeneous Database Architecture

David M. Hansen
George Fox University, dhansen@georgefox.edu

An Object-Oriented Heterogeneous Database Architecture

David Marshall Hansen
B.S. Computer Science, Oral Roberts University, 1984
M.S. Computer Science, Washington State University, 1988

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 1995

The dissertation "An Object-Oriented Heterogeneous Database Architecture" by David Marshall Hansen has been examined and approved by the following Examination Committee:

David Maier
Professor
Thesis Research Adviser

Lois Delcambre
Associate Professor

Jonathan Walpole
Associate Professor

James Stanley
Assistant Professor
Materials Science Department

Dedication

To my father, Richard K. Hansen,
who taught me to worship God and enjoy exploring His boundless creation;
and to my son, Richard K. Hansen,
for whom I hope to do the same.

Acknowledgements

First of all, thanks to my advisor, David Maier, for seeing me through the many difficulties of graduate school—I came to OGI hoping for the opportunity to work with you and it has been a greater privilege than I imagined. Thanks to Jim Stanley for his patience with this pseudo-scientist (who still can't figure out why it's *h, k, l* instead of something sensible like *i, j, k*). Thanks also to the other members of my committee, Lois Delcambre and Jon Walpole, for their advice and patience—together they turned writing this dissertation into a valuable learning experience.

Thanks to my friends and colleagues at Battelle, Pacific Northwest Laboratories for the time and financial support to pursue this degree—especially Paula and Jim for your mentoring and encouragement throughout my career.

Thanks Ken for the tennis, Jeff et al. for the volleyball, Eric for the golf, Roger for the fishing, the OGI "music department" for the cultural enrichment, and the OGI softball team for the laughs. You all helped keep me healthy and sane.

Finally, many thanks to my wife, Paulette, for her love, patience, and support—emotionally and financially. I could *never* have done this without you!

A wife of noble character who can find?
She is worth far more than rubies—Prov. 31:10

Through the years, my research has been supported by the Oregon Graduate Institute, a fellowship from the Oregon Advanced Computing Institute (OACIS), and NSF Grant IRI-9117008.

Contents

List of Tables

List of Figures

List of Pseudo-Code Fragments

List of SmalltalkDB Fragments

List of OODBMS Fragments

List of C Programs

Abstract

An
Object-Oriented
Heterogeneous Database Architecture

David Marshall Hansen, Ph.D.
Oregon Graduate Institute of Science & Technology, 1995

Supervising Professor: David Maier

Many data management environments face a critical need to integrate heterogeneous data—data that are stored in varying locations using various data management systems with diverse data formats and schemas. To address this problem, the database research community has developed the concept of a heterogeneous database system (HDB) that provides users with the illusion of a single unified database. However, HDBs rely on the implicit assumption that all data to be integrated into the HDB are stored in full-fledged database management systems (DBMS). This assumption leaves environments that need to integrate non-DBMS data unserved by HDB systems. Furthermore, HDBs are complex software solutions that are not easily implementable by database developers wrestling with heterogeneous data. This thesis presents a new, easily implemented HDB architecture that is suitable for integrating non-DBMS data.

The key to our architecture is using an object-oriented database management system (OODBMS) as an implementation tool. Rather than developing an HDB from scratch, we leverage the power and facilities of the underlying OODBMS to provide a query language, application programmer interface, interactive query interface, concurrency control, etc.

Using object-oriented technology gives us an additional benefit—our HDB becomes an object-oriented HDB (OOHDB) providing users with greater data model expressivity along with a powerful behavioral component.

The OOHDB architecture we present is independent of a particular OODBMS and can be implemented using a number of commercial OODBMSs for a variety of data management environments. We describe one implementation of our architecture using the GemStone OODBMS for accessing heterogeneous materials science data. This implementation demonstrates how easily the architecture can be implemented. We use this implementation to analyze the performance of the architecture and examine the effectiveness of strategies for enhancing performance.

We conclude that for many environments with heterogeneous non-DBMS data, our OOHDB architecture provides a good solution that is easy to implement using commercial OODBMS technology.

Chapter 1

Introduction

1.1 Overview

Many data management environments face a critical need to integrate heterogeneous data—data that are stored in varying locations using various data management systems with diverse data formats and schemas. To address this problem, the database research community has developed the concept of a heterogeneous database system (HDB) that provides users with the illusion of a single unified database [Kim95a, Ram91, SL90, LMR90, LA86, EP90]. However, proposed HDB architectures implicitly assume that all data to be integrated into the HDB are stored in full-fledged database management systems (DBMS). This assumption leaves environments that need to integrate non-DBMS data unserved by HDB systems. Furthermore, HDBs are complex software solutions that are not easily implementable by database developers wrestling with heterogeneous data. This thesis presents a new, easily implemented HDB architecture that is suitable for integrating non-DBMS data.

1.2 What is a Heterogeneous Data Base?

Database researcher Won Kim provides the following succinct description of what an HDB should be and do (referring to the concept as a multidatabase system (MDBS)):

> Simply put, a multidatabase system (MDBS) is a database system that resides unobtrusively on top of existing database and file systems (called *local database systems*) and presents a single global database schema against which

> its users will issue queries and updates; an MDBS maintains only the global schema, and the local database systems actually maintain all user data. The global schema is constructed by consolidating (integrating) the schemas of the local databases;...The MDBS translates the global queries and updates for dispatch to appropriate local database systems for actual processing, merges the results from them, and generates the final result for the user [Kim95a, p.516].

Kim goes on to list 9 general objectives of an HDB (or MDBS) [Kim95a, pp.516–517]:

OBJECTIVE 1 It must obviate the need for a batch conversion and migration of data from one data source (e.g., an ORACLE database) to another (e.g., a Sybase database).

OBJECTIVE 2 It must require absolutely no changes to the local database system (LDBS) software; this preserves what is known as *design autonomy.* In other words, an MDBS must appear to any of the LDBSs as just another application user.

OBJECTIVE 3 It must not prevent any of the LDBSs from being used in its native mode. In other words, users of an LDBS may continue to work with the system for transactions that require access only to data managed by the systems, while users will use the MDBS to issue transactions that require access to more than one data source. In this way, applications written in any of the LDBSs are preserved, and new applications that require access to more than one data source may be developed using the MDBS.

OBJECTIVE 4 It must make it possible for users and applications to interact with it in one database language. In other words, the users and applications should not have to work with the different interface languages of the LDBSs.

OBJECTIVE 5 It must shield the users and applications from the heterogeneity of the operating environments of the LDBSs, including the computer, operating system, and network protocol.

OBJECTIVE 6 It, unlike most previous attempts at allowing the interoperability of heterogeneous database systems, must support distributed transactions involving both reads and updates against different databases.

OBJECTIVE 7 It must be a full-blown database system—that is, it must make available to users all the facilities provided by standard database systems, including schema definition, non-procedural queries, automatic query optimization, updates, transaction management, concurrency control and recovery, integrity control, access authorization, both interactive and host-language application support, graphics application development tools, and so forth.

OBJECTIVE 8 It must introduce virtually no changes in the operation and administration of any of the LDBSs.

OBJECTIVE 9 It must provide run-time performance that approaches that of a homogeneous distributed database system.

Two points regarding Kim's description and objectives must be made. First, the description and objectives are prescriptive. That is, they are an ambitious list of capabilities that an HDB *should* strive to achieve to be a fully-functional, non-intrusive solution. In fact, while most of Kim's objectives are met by current HDB approaches, objectives dealing with updating heterogeneous external data via the HDB (objectives 6 and 7) remain a topic of research. Propagating updates to external data via an HDB is difficult because it not only requires a mechanism, but an invertible mapping from elements of the heterogeneous schemas to elements of the homogeneous schema as well. Thus, HDBs typically do *not* provide an update capability.

Second, though Kim's description mentions sitting atop "existing database and *file systems*...", data that is not stored in a DBMS is rarely integrated into an HDB. Close examination of Kim's objectives makes it clear that the objectives are biased toward describing an HDB that accesses data managed by DBMSs. To begin with, Kim refers to external sources as "local *database systems*". Objectives 2, 3, 6, and 7 deal with transactional issues that are not typically relevant for files. Objective 4 mentions the

"interface language" of the local databases. These subtle hints suggest that HDBs are oriented towards supporting the integration of DBMS data and non-DBMS data is largely ignored. As we shall see in Chapter 2, the prototypical HDB architecture, where queries over the global HDB schema are translated into sub-queries that are passed along to local databases for execution, all but excludes data that is not managed by a powerful DBMS.

1.3 What is the Problem with HDBs

We see two problems with current HDB approaches. First, the implicit assumption that all external databases[1] are managed using DBMSs makes these approaches unsuitable for environments with data that is not stored in a DBMS. Kim's description notwithstanding, his list of objectives is heavily biased towards integrating DBMS data and the fact is that most HDB architectures provide little or no support for accessing non-DBMS data. Second, and more generally, HDB systems are complex one-of-a-kind software solutions that are not easily implemented by database developers wishing to integrate their heterogeneous databases.

1.3.1 "Lightly-Managed" Data

One drawback of Kim's objectives, and virtually all HDB research, is that it presumes that all external databases are managed using powerful general-purpose DBMSs. Furthermore, the assumption is that these databases are relational [BHP92]. However, many environments have "databases" that are *not* managed by a general-purpose DBMS including: defense [AMR94], medicine [WH94], telecommunications [CD93], geophysics [DSH94], molecular biology and genomics [Kar94, Ald93, SR94], chemistry [RL85], and materials science [HS91]. "Databases" in these environments are often formatted files containing large data sets that may include historical data (e.g., telephone customer records, chemistry experiment records) or databases of factual information (e.g., the map of a gene fragment, physical properties of a material). The data in these sorts of databases remain

[1] We use the term *external database* as a synonym for *local database* from here on because it is more accurate. The term *local* incorrectly connotes that the databases are co-located with the HDB. However, in practice, databases accessed by an HDB are often distributed, sometimes widely.

static once collected.

We term these databases "lightly-managed". The common characteristic of these lightly-managed databases is that they lack a powerful, general-purpose query interface. Instead, custom data access programs provide limited access to the data along pre-defined access paths. Without a general-purpose query access mechanism, lightly-managed databases remain outside the realm of current HDB approaches.

When dealing with lightly-managed data, some of Kim's objectives take on less significance, others more. Of the objectives listed, those that deal with the "design autonomy" of the local DBMS and the coordination of distributed transactions (objectives 2, 3, 6, and 8) become largely irrelevant in the absence of a DBMS. Objectives that define the level of transparency and performance (objectives 1, 4, 5 and 9) remain important, while objective 7—specifying that the HDB should be a full-fledged DBMS—takes on additional importance since it implies that the HDB should provide even greater query and management capabilities than those of the lightly-managed database. In essence, an HDB that includes a lightly-managed database should expand the capabilities for users of that database.

1.3.2 Keeping it Simple

Another drawback to most HDB systems is that they are very complex pieces of software. Decomposing and optimizing a global query across the external databases is a complex task in itself. The architecture of Pegasus, a heterogeneous information management system from researchers at Hewlett-Packard Laboratories, is presented in Figure 1.1. Pegasus is representative of the complexity of most HDB solutions.

As complex, customized pieces of software, each HDB solution is a one-of-a-kind system that may be tailored for a particular environment. As of yet, HDBs remain a topic of research. M.W. Bright et al., surveying current multidatabase systems, note that of the 16 HDBs surveyed, 13 are "prototypes" and the other 3 are "research" systems [BHP92, p.56]. HDB research may eventually lead to a general-purpose HDB that is customizable for a particular environment. However, as long as HDB systems continue to ignore lightly-managed data, users with such data will find it a daunting task to develop or modify an HDB to suit their needs.

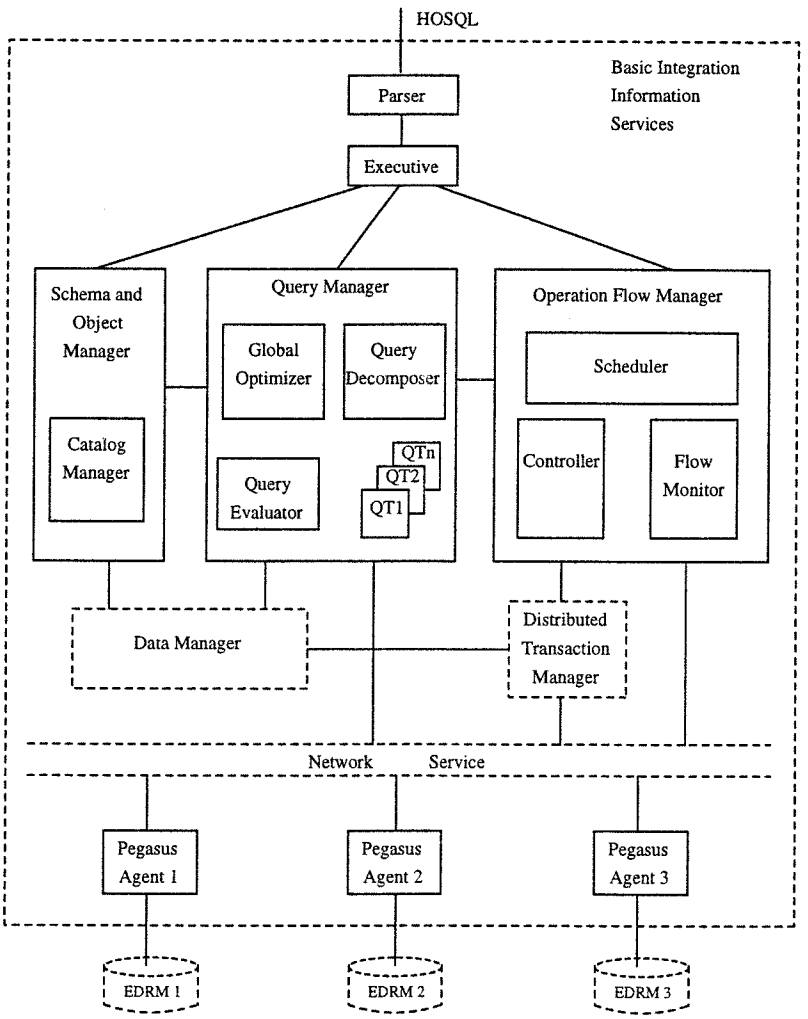


Figure 1.1: Pegasus Component Structure [SAD+95, p.669]

1.4 Our Thesis

Our thesis is two-fold. First, we believe that an HDB can *provide optimized access to lightly-managed databases.* What is needed is a new HDB approach to integrating external databases that does not assume them to be managed by a DBMS.

Second, we believe that a general-purpose high-level HDB architecture can be specified that is *easy to implement.* Ease of implementation has two facets. First, powerful tools for implementing the HDB must be readily available. Second, the implementation must be simple, straightforward, and must not require significant code development. The

reader might reasonably ask how we hope to achieve much simplification in the development of such a complex software system. The answer lies mainly in our decision to use object-oriented database management system (OODBMS) technology for building an HDB. Much of the complexity involved in programming an HDB is in the task of building the components that translate, decompose and optimize a global query across external databases. Our approach uses the query language and query processing engine of the underlying OODBMS instead of developing a custom query engine. In fact, by building an HDB using a DBMS as the *implementation tool* we leverage the features of the DBMS to build an HDB with *all* the "full-blown" DBMS features outlined by Kim in objective 7. The HDB "inherits" the query language, concurrency mechanisms, application programmer interface, etc. of the underlying OODBMS. Furthermore, using object-oriented technology to construct our HDB provides the added benefit of turning the HDB into an object-oriented HDB (OOHDB). The power of the object-oriented paradigm is especially significant when applying the HDB in domains where complex data models have hindered the use of traditional DBMS technology, such as scientific data management [FJP90]. An object-oriented data model provides a rich and powerful mechanism that can be used to model data that is not easily decomposed into the rows and columns of the relational model. An object-oriented data model can describe the domain using entities intuitively understood by the users of the database.

1.5 Research Methodology

Chronologically, our research was conducted in three phases. We began by identifying a suitable real-world domain for developing and testing an OOHDB. Next, we implemented an OOHDB that provides users and their application programs with access to a number of heterogeneous lightly-managed databases. Finally, a high-level generalized OOHDB architecture was identified that can be applied in a variety of domains and implemented using commercial OODBMS products.

1.5.1 The Test Domain—Materials Science & Crystallography

Materials science provided a useful domain for implementing and testing an OOHDB. Materials scientists rely extensively on programs and computerized data for conducting research [Hal85]. Much of the data relevant to materials science is contained in lightly-managed databases. In particular, published commercial data sets covering the crystallographic structure of many materials are available from standards organizations. Data stored using standardized data-interchange formats designed for crystallographers provides another lightly-managed data source.

1.5.2 Implement an OOHDB for Materials Science

We implemented an OOHDB for crystallographic databases using a commercial OODBMS. Our decision to use a "commercial-off-the-shelf" (COTS) OODBMS is an important feature of our research. HDB systems are too often constructed using custom software systems that make the technology inaccessible to database developers. We choose to use COTS systems to demonstrate that powerful tools for building an OOHDB are readily available.

This implementation was used to develop and test general techniques for HDB construction and optimization. This implementation was carefully crafted to avoid using the proprietary features of a particular OODBMS so that the architecture could be implemented using a number of commercial OODBMSs.

1.5.3 Generalize a High-Level Architecture

Finally, we identified and developed the specification for an OOHDB architecture that is independent of any particular data management domain or OODBMS. This architecture is generalized from the implementation of the our OOHDB for materials science and embodies a general methodology for constructing an OOHDB using a COTS OODBMS. The architecture was examined in the light of three COTS OODBMS products to assess its generality.

1.6 Contributions of Research

The principal contribution of our research is the development of a high-level OOHDB architecture. We believe that the architecture is:

- Suitable for integrating lightly-managed databases.
- Simple and easy to implement using a variety of COTS OODBMSs

1.7 Limitations

Our research is not without limitations, however, both in scope and applicability. The scope of the research is confined to the development of an OOHDB architecture. HDB research has a number of interesting problems we have not set out to solve including:

Tools and techniques for integrating heterogeneous schemas — Though we developed a global schema for our test domain, this thesis presents no new tools or techniques for schema integration. We give examples of how we use the computational power of methods in the OODBMS to handle instances of syntactic and semantic heterogeneity among external databases, but in general, the problem of homogenizing heterogeneous schemas and data remains.

Updates and distributed transactions — As we noted previously, propagating updates to heterogeneous databases via the OOHDB is an open research topic. Our architecture does not attempt to solve this difficult problem. However, we do suspect that the underlying transaction management capabilities of the OODBMS can be used to implement distributed, heterogeneous transactions in an environment that requires and supports them. Furthermore, in Chapter 6 we outline a mechanism for extending our architecture to propagate updates. Of course the mechanism does not solve the difficult problem of developing an invertible mapping from elements of the external database schemas to elements of the homogeneous schema of the OOHDB.

Decomposing and optimizing global queries — Our architecture is quite different from traditional HDB approaches. Thus the common problem of decomposing and

optimizing global queries is not addressed. However, query optimization within our OOHDB *is* an important consideration, and performance enhancing optimizations are discussed as a part of the architecture.

At its current stage, our work is also somewhat limited in its applicability. The architecture we have developed trades real-time currency of the data in the OOHDB for enhanced query performance. Specifically, a static representation of the data in external databases is constructed within the OODBMS. Updates, deletions, and insertions to external databases are *not* immediately visible to users of the OOHDB. The representation within the OOHDB must be updated in order for external database modifications to become visible. Thus, our architecture is probably not a suitable choice for transaction-processing environments where external databases are in a constant state of change.

1.8 Outline of the Thesis

The remainder of this thesis follows a logical rather than chronological organization:

Chapter 2 examines approaches to accessing lightly-managed databases. Many of these approaches fall far short of providing a true HDB. We also survey other HDB research and demonstrate that lightly-managed data is not well served by current HDB approaches.

Chapter 3 presents our high-level domain-independent OOHDB architecture. We describe how we will leverage the power of the underlying OODBMS to simplify the implementation while still providing a full-fledged OOHDB. By using the database features of the underlying OODBMS, most of the complexity in constructing an HDB is eliminated. We evaluate our architecture using Won Kim's objectives discussed in Section 1.2.

Chapter 4 presents a detailed description the implementation of our OOHDB for a materials science crystallographic database. This chapter demonstrates that the architecture can be implemented to solve a real-world problem and provides evidence for our claim that the architecture can be easily implemented using COTS OODBMS

technology. We analyze the performance of the materials science OOHDB and examine the effectiveness of strategies for enhancing performance.

Chapter 5 strengthens our claim that our architecture is OODBMS-independent by examining two other COTS OODBMS products to assess their suitability as targets for implementing our OOHDB. Where a particular OODBMS would require changes to the implementation described in Chapter 4, we present alternative implementation strategies.

Chapter 6 revisits our thesis to assess the simplicity and suitability of our architecture for lightly-managed databases. We reiterate the lessons we have learned along the way and discuss some directions for future research.

Chapter 2

Related Research

Our discussion of related research examines work by others building HDBs as well as non-HDB approaches to accessing lightly-managed data. While HDB approaches do a poor job of integrating lightly-managed databases, non-HDB approaches make no attempt to provide the transparent integration of an HDB.

2.1 Querying and Accessing Lightly-Managed Data

Most approaches to querying and accessing lightly-managed databases make no attempt to provide the transparent integrated interface of an HDB.

2.1.1 Querying Files

Front-ends for querying data stored in operating system files have been around for quite some time. The commercial product Datatrieve[1] is typical of this genre of software solutions.

Datatrieve adds high-level query capabilities to operating system files. Highly-structured record-oriented files are described using a Cobol-like schema syntax. Datatrieve provides users and application programs with an SQL-like high-level query language for expressing queries over data stored in operating system files. Access can be optimized by constructing indexes. Two limitations to Datatrieve are that the data files need to follow a *very* rigid record-oriented structure and second, although multiple files can be accessed simultaneously, there is no capability for homogenizing heterogeneous data sources.

[1] *Datatrieve* is a registered trademark of Digital Equipment Corporation

2.1.2 VISTA—A Metadata Approach

Another common approach to providing access to lightly-managed data is the "metadata" approach typified by the VISTA System [DSH94]. The VISTA (Visual Interface for Space and Terrestrial Analysis) System provides a visual query interface to large geophysical data sets. VISTA provides the user with a database of metadata—data *about* the underlying data sets, such as date and time of collection, location, and general features. Users of VISTA query the metadata to locate data sets of interest. The data sets themselves are stored using any number of non-DBMS data-interchange formats such as the Hierarchical Data Format (HDF), Flexible Image Transport System (FITS), Network Common Data Format (netCDF), etc. Once selected, a data set may be displayed and manipulated using the VISTA System.

The approach taken by VISTA differs from our approach in a number of ways. First, VISTA is clearly a domain-specific system. VISTA understands data-interchange formats that are commonly used by scientists and is designed to display and manipulate geophysical data sets. Second, VISTA uses a limited schema of metadata that does not allow a user to query over all the attributes of the data. This approach is similar to other "directory" approaches where the database is primarily used as a directory manager for locating the *real* data. Third, VISTA is useful for locating data sets, but does not attempt to provide users with a homogeneous view of the data. Finally, VISTA is an end-user system and provides no general-purpose application programmer interface (API) for user applications. VISTA can only pass selected data sets along to other data analysis software packages.

2.1.3 The Aurora Dataserver—An Extended Relational Approach

The Aurora Dataserver[2] for visualization applications [XID94, Jir93] also provides a metadata approach, but takes the approach a step farther by integrating lightly-managed databases as "dataset" values in an extended relational data model. The Aurora Dataserver is built atop the Orion[3] extended-relational database management system. The dataserver

[2] *Aurora Dataserver* is a trademark of XIDAK Inc., Palo Alto, CA
[3] *Orion* is a trademark of XIDAK Inc., Palo Alto, CA

is targeted at providing relational data management for scientific domains with large "n-dimensional coordinate data". The dataserver extends the relational model by defining a new *dataset* type that provides bulk storage of data sets. The dataserver also defines a set of operations for manipulating *dataset* values. Thus, while a query in VISTA returns pointers to help the user locate the real data sets, an Aurora Dataserver query is capable of returning the data sets themselves. Figure 2.1 depicts the Aurora Dataserver architecture. The dataserver provides users and their applications with interfaces to the database typical of a relational database management system. User-developed importer

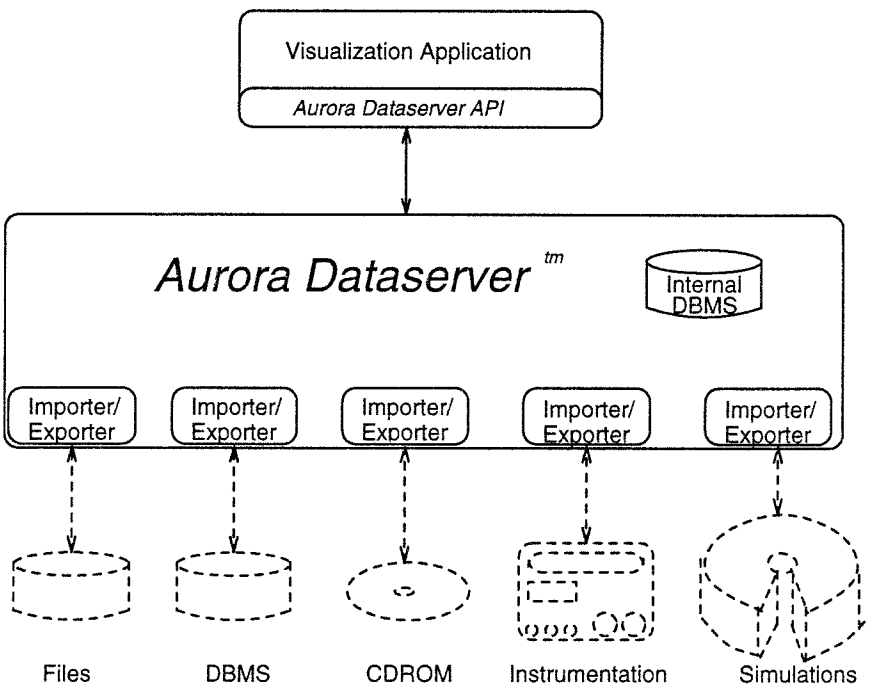


Figure 2.1: The Aurora Dataserver Architecture [Jir93, p.6]

and exporter functions are used by the dataserver to access external data. External data sets are homogenized by converting them into instances of the Aurora Dataserver *dataset* type. Files of external data can either be imported or "registered" with the dataserver. Files that are imported are copied into Aurora's database and converted into *datasets*, allowing the dataserver to optimize future data access. Registered files *appear* as *dataset* values stored by the dataserver, but actually remain in their original location to minimize data storage requirements.

Although the Aurora Dataserver is a powerful tool for manipulating scientific data sets, it is not versatile enough to be a true HDB. The narrowly defined *dataset* type used by the Aurora Dataserver is only suitable for domains with data sets that are n-dimensional coordinate data (e.g., matrices).

2.2 HDB Research

Heterogeneous database systems have been an on-going topic of research for many years. HDB research is often divided into those systems that provide "interoperability" among external databases (no global schema), and those that provide "integrated" access to external databases through a global schema. Our research is interested in the more transparent integrated solutions, so our discussion here is confined to integrated HDB approaches.

One of the earliest integrated HDB prototypes was the Multibase system [SBD+81]. The simple diagram of Multibase shown in Figure 2.2 has influenced the direction of most subsequent HDB research. The basic function of Multibase is to maintain a global schema

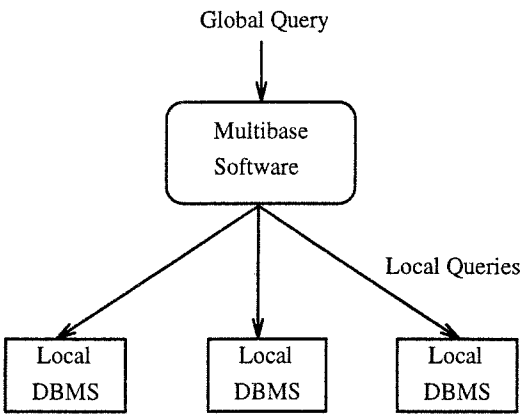


Figure 2.2: The Multibase Architecture [SBD+81, p.336]

and translate queries against that schema into queries over external databases (called "local DBMS" by Multibase). An interesting consequence of this simple architecture is that most HDB research presumes that external databases are managed by powerful DBMSs. This assumption has resulted in HDB architectures that integrate lightly-managed databases only as an after-thought if at all, and then not very well. Furthermore, the reliance on a relational model for external DBMSs has become so ingrained in the research that

one researcher has suggested that the relational data model should simply be adopted as a standard, canonical model for building HDB systems [LA86].

2.3 Object-Oriented HDB Research

The notion of building an HDB presenting an object-oriented data model is relatively new [Man89, SCGS91, CT91, LM91, Ber91, TSB92]. In some cases, the use of object-oriented technology is for infrastructure only—the HDB appears relational to its users [LM91]. In other cases, the HDB presents its users with an object-oriented data model yielding a true OOHDB [CT91, Ber91, KDN90]. These OOHDB efforts are intended to exploit the expressivity of the object-oriented data model to model complex objects.

One common strategy for homogenizing heterogeneous data using an OOHDB is the view mechanism proposed by Bertino [Ber91]. The notion is that abstract classes, or views, can be used to provide a homogenizing layer atop heterogeneous classes and objects.

Kaul et al. [KDN90] also propose an object-oriented view mechanism for integrating data. Furthermore, they describe a prototype implementation using Smalltalk called the ViewSystem. "External" classes of objects can be defined in the ViewSystem that are "non-materialized". Queries over external classes result in a materialized collection of objects that satisfy the query. The class hierarchy is used to group similar external and internal class definitions together using a common superclass to provide a homogeneous view of data.

The ViewSystem makes an attempt to integrate lightly-managed databases, however, the mechanism used is primitive—importing files in their entirety on demand. Nonetheless, this is one of the few attempts to address lightly-managed data in HDB research.

Both Kaul et al. and Bertino use a hierarchy of class definitions to homogenize heterogeneous data. However, we believe that the use of the class hierarchy as a mechanism for homogenization has serious drawbacks and in Chapter 3 we will contrast this approach to homogenization with our own.

Even more germane to our research, a few researchers are exploring the use of an OODBMS as a *tool* for constructing an OOHDB [SAD+95, CL88, RD94, HMZ90].

2.3.1 Building an OOHDB Using an OODBMS

Connors and Lyngbaek [CL88] appear to be the first to propose the use of an OODBMS as a tool for constructing an HDB. They propose the use of the Iris[4] OODBMS from Hewlett-Packard Laboratories to construct a *global data manager* providing uniform access to heterogeneous data. In choosing to use OODBMS technology as an implementation tool, they note:

> The extensible nature of an OODBMS, i.e., the provision for abstract data types and operations, makes it feasible to write interfaces to a wide variety of existing information sources and that way create the illusion of a single integrated database which can be queried in a uniform manner [CL88, p.162].

In other words, by leveraging the behavioral component of an OODBMS, the OODBMS can be used as a tool for constructing an OOHDB.

They propose to access external data via "foreign functions" that retrieve data on demand. For example, using a stock market database they suggest that the foreign function *MarketPrice*, taking a stock symbol as input, could return the current market value of a stock by interacting with an on-line quote service. Connors and Lyngbaek make an important observation regarding this functional approach to integration:

> The approach described in this paper has a procedural flavor. By using information-importing functions, it is not necessary to globally conform the local schemas. Rather, the programmer defining an information-importing function explicitly specifies a procedure that implements a mapping from the external information of interest to the importing database [CL88, p.164].

In essence, their approach achieves homogeneity by explicitly mapping heterogeneous data to a homogeneous form during importation. This contrasts with the class-hierarchy approach to homogenization proposed separately by Kaul et al. and Bertino.

However, the foreign function approach to importing data does not appear to be particularly OODBMS-independent. While they suggest that their approach could be supported

[4]Iris is now available commercially as *OpenODB* from Hewlett-Packard

by other OODBMS with general-purpose programming languages, they conclude that:

> Iris' support for generalized query processing capabilities and operations written in arbitrary programming languages make it a better candidate for a global data manager than many other OODBMSs [CL88, p.172].

However, we do not believe that "operations" in the form of the foreign functions used by Iris are all that convenient or user-friendly. Foreign functions must be compiled and carefully linked into the Iris query processing engine in order to be accessible. The process as outlined here is clearly not for the casual Iris user. Furthermore, Iris is a functional OODBMS that is object-oriented mostly in the sense that it allows for user-defined data types. Generalized computational behavior in the form of methods is absent from the Iris data model and must be added through the use of linked-in functions written in "arbitrary" programming languages.

The work of Connors and Lyngbaek has a number of similarities to our own. Specifically, both use the power of the underlying OODBMS to provide the basic features of an OOHDB. Both use a "procedural approach" to schema integration. However, where they use foreign functions linked into the Iris query processing engine to access external data, we use nothing more difficult to master than the data manipulation and method definition language of the underlying OODBMS. We believe that this makes our approach both simpler and more general. Furthermore, our work presents an approach for integrating and optimizing data stored in lightly-managed databases, a capability that the global data manager may possess, but that remains unexplored.

2.3.2 The Pegasus OOHDB

Another OOHDB project developed using Hewlett-Packard's OpenODB product is the Pegasus system [Sha93, SAD+95]. However, in contrast to the global data manager of Connors and Lyngbaek, Pegasus represents a much more traditional HDB approach that relies less on the native power of the underlying OODBMS than on a complex software architecture (see Figure 1.1). With its reliance on powerful external DBMSs, the Pegasus

system is typical of most HDB systems that are not well-suited to integrating lightly-managed databases.

2.3.3 An OOHDB for Molecular Biology

Using the commercial OODBMS ObjectStore[5], Rieche and Dittrich have implemented an OOHDB for molecular biology that provides access to lightly-managed molecular biology data [RD94]. Rieche and Dittrich describe a very pragmatic and domain-dependent approach to data integration that makes no pretense about trying to "invent new concepts for federated database systems." The mechanism for querying lightly-managed data is quite unique—global queries are transformed into programs that scan files for relevant data. Files containing any relevant data are loaded in their entirety into the OOHDB.

This sort of brute-force file query mechanism demonstrates a common problem when integrating lightly-managed databases into an HDB. Since lightly-managed databases typically lack a query facility, either a query facility must be developed for files (as Rieche and Dittrich have done), or the files must be brought into the HDB where a query facility can be applied. Our architecture presents a simple and novel solution to this problem. Our solution uses the query processing engine of the underlying OODBMS without loading lightly managed databases entirely into the OOHDB.

2.4 Summary

Clearly, there are workable non-DBMS approaches capable of providing access to lightly-managed data. But these approaches do not provide the benefits of an HDB—most notably, they lack transparent integration of heterogeneous data.

Previous HDB research has concentrated primarily on providing access to data managed by DBMSs and important sub-issues such as global query translation and optimization, transaction management, schema integration, etc. We recognize that these continue to be complex and difficult problems requiring solutions.

Research into the use of object-oriented technology for constructing an OOHDB is

[5] *ObjectStore* is a trademark of Object Design, Inc.

providing both strategies for modeling heterogeneous data, as well as mechanisms for constructing an OOHDB using an OODBMS. Approaches such as the global data manager suggest that leveraging the power of an OODBMS system—in effect using a database management system to *build* a database management system—is a reasonable approach. In fact, we believe it is an approach that makes it much easier to build an OOHDB. Still, integrating lightly-managed databases remain an after-thought, and common brute-force integration approaches seem overly complex and inefficient.

We believe that building an OOHDB that integrates lightly-managed databases requires a new approach. This thesis presents a new OOHDB architecture that provides the efficient access to lightly-managed databases of the VISTA system while providing the transparency and rich object-oriented data model of the Pegasus OOHDB.

Chapter 3

An Object-Oriented HDB Architecture

We have set two goals for our OOHDB architecture: the OOHDB should be easy to implement and the OOHDB should be suitable for integrating lightly-managed databases.

In order to achieve a powerful OOHDB that is simple to implement, we rely on OODBMS technology as an implementation tool. Implementing an OOHDB thus becomes a matter of harnessing the capabilities of the OODBMS rather than developing the capabilities ourselves. The architecture we present here presumes a relatively common object-oriented model based on classes of objects that encapsulate state and behavior.

The overall approach we use to integrate data is quite simple. As shown in Figure 3.1, The OOHDB is populated with objects—one for each entity identified in each external data source. These objects store no data within the OOHDB. Instead, when the object needs some data (e.g., in response to a query), the object requests the data from another object within the OOHDB that encapsulates all access to a particular data source. This "database encapsulating object" retrieves and converts the required data from the external data source. This approach allows us to integrate lightly-managed databases because it does not require their data to be accessible via some powerful query processing facility. All that we require is to be able to selectively retrieve an entity from a data source. Section 3.1.3 will further clarify our proposal for integrating lightly-managed databases.

The most common alternative to our approach of populating the OOHDB with individual objects is to represent entire collections of external data entities with a *single* "pseudo-collection" object in the OOHDB. When queried, this pseudo-collection object translates the query into sub-queries over the external databases it represents. From our perspective, the two problems with this approach are that it falls into the common trap

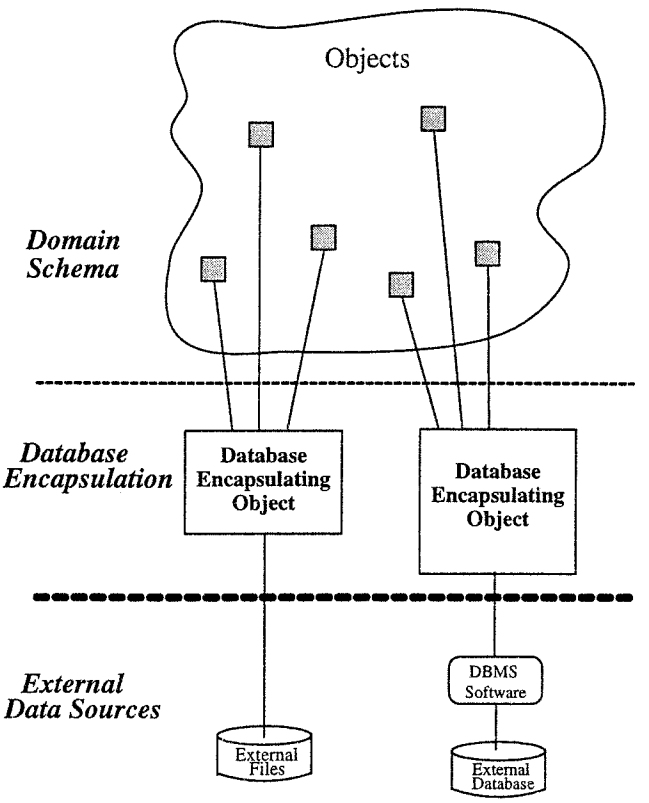


Figure 3.1: Integrating External Data

of requiring external databases to be managed by a powerful DBMS and it requires substantial development to build robust query translators. Since we are intent on integrating lightly-managed databases, we have developed an approach that minimizes the required functionality of the external data source and uses the underlying OODBMS for query processing and optimization.

Our approach is not without its limitations. The most obvious limitation is that building a static representation of external data within the OOHDB means the OOHDB may not always be up-to-date with respect to the external data. This static representation may limit the applicability of our architecture where modifications to the external data are very frequent such as on-line transaction-processing environments. In evaluating the applicability of our approach to a particular environment, the important questions are whether the currency of the OOHDB can be maintained efficiently, and whether users of the OOHDB can tolerate data that may be out-of-date. The domains mentioned in

Section 1.3.1 typically include large, slowly-changing databases that can benefit from our approach.

A second limitation is that we expect use of the OOHDB to follow a predictable pattern of access. The expected pattern is one where users will query large collections in the OOHDB using common paths that we can optimize. This access pattern is typical of the pattern that metadata approaches such as VISTA expect and accommodate for scientific data. Queries using optimized paths will not require any access to external data and new paths can be supported at any time. As we shall see in Chapter 4, accessing external data is very time-consuming and must be minimized if the OOHDB is to achieve reasonable performance.

3.1 Our Layered Architecture

The key feature of our architecture is the method we propose for connecting the objects in the OOHDB with the external data sources that contain their data. As Figure 3.1 depicted, the architecture consists of three layers. The Schema Layer at the top of our architecture is comprised of the objects defined by the user's domain schema. These are the objects that users of the OOHDB query. The Database Encapsulation Object Layer in the middle of the architecture contains "database encapsulating objects" (DEOs). Each DEO encapsulates all access to a particular external data source. Finally, the External Data Source Layer at the bottom of the architecture consists of the data sources themselves.

3.1.1 The Schema Layer

The purpose of the Schema Layer is to hide both the format and location heterogeneity of external data. That is, users and their applications that query and manipulate the objects of the domain schema should not be aware of the fact that the data is stored externally in heterogeneous databases. There are two object-oriented approaches to hiding heterogeneity.

One approach is to use the object-class hierarchy to mask heterogeneity. This is similar to the "view"-oriented approaches of Kaul et al. and Bertino mentioned in Section 2.3.

The idea is to use a common superclass to provide a "view" of a number of heterogeneous classes. The superclass provides a homogenized interface and definition for the heterogeneous classes. For example, in Figure 3.2, the class *Crystal* provides a homogenized interface to heterogeneous subclasses that draw data from different data sources. Each external data source is encapsulated using an object that provides a homogeneous method interface (above the dashed-line) to heterogeneous attributes (below the dashed-line).

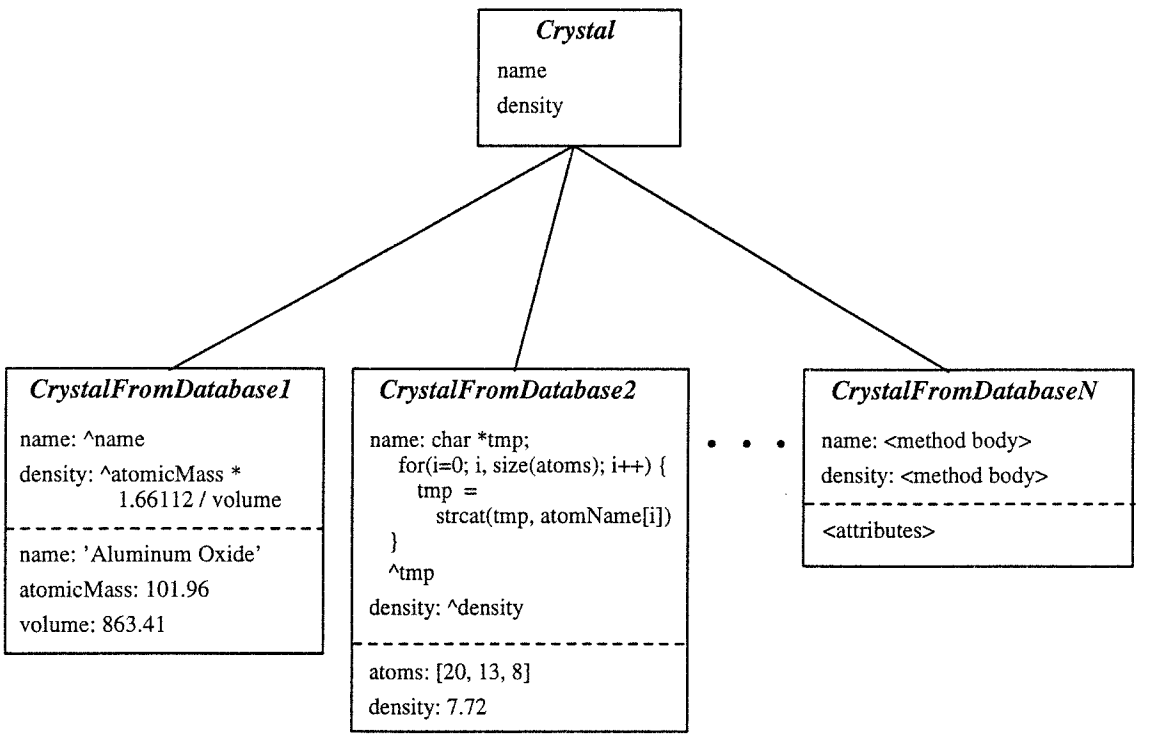


Figure 3.2: Homogenization Using the Class Hierarchy

The problem with this approach is that it uses the class hierarchy to hide implementation details. For example, if we wish to add *InorganicCrystal* as a subclass of *Crystal*, *InorganicCrystal* must have its own hierarchy for masking heterogeneity as shown in Figure 3.3. The class hierarchy quickly becomes muddled, unwieldy, duplicative, and confused—a maintenance nightmare. Clearly the relationship between *Crystal* and *InorganicCrystal* is of a different sort than the relationship between *Crystal* and *CrystalFromDatabase1*. A class hierarchy is insufficient for distinguishing the difference and mixing the two is problematic.

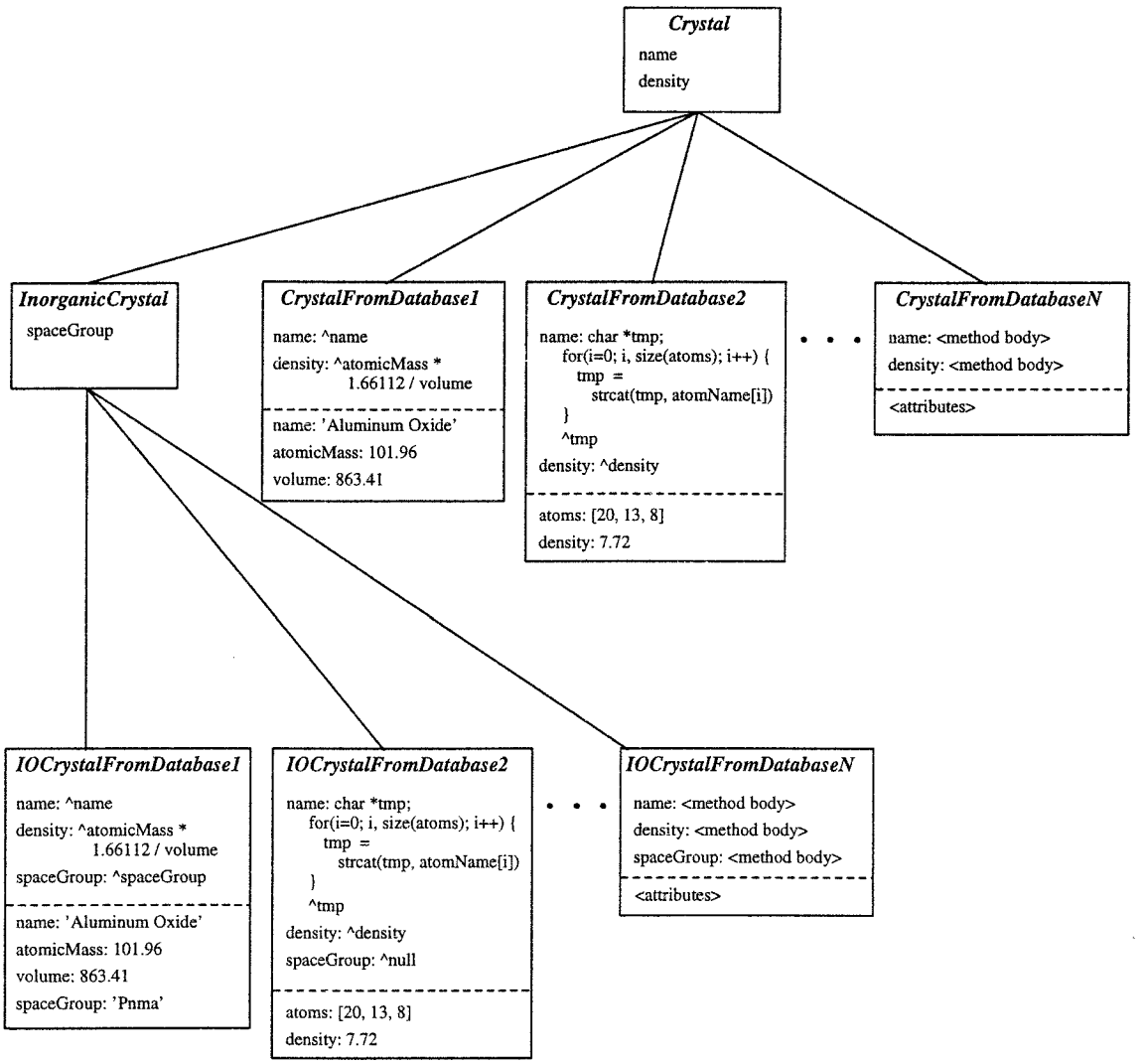
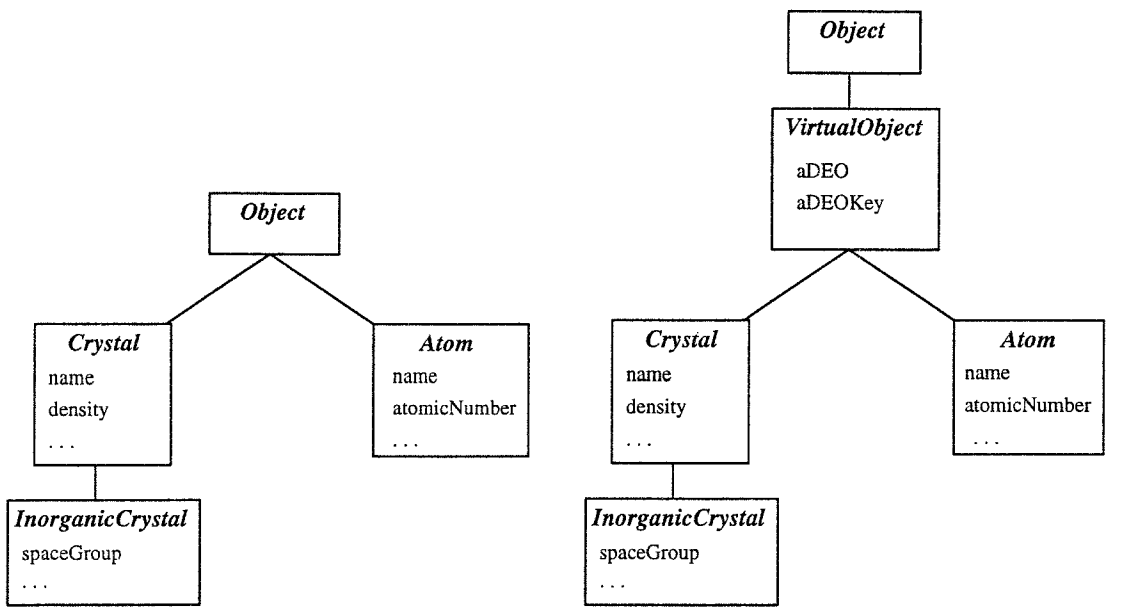


Figure 3.3: A Class Hierarchy Hiding Implementation Details

We take a different approach. Instead of using the class hierarchy to mask heterogeneity, we separate heterogeneous data source details from the inheritance hierarchy. We begin by specifying a new "root" class from which most classes will inherit. In a typical object-class hierarchy, each hierarchy is rooted in a common base class. In Figure 3.4, we introduce a new root class, *VirtualObject*, with two attributes: a reference to a DEO, and a key for this object that will be used by the DEO to retrieve the object's attribute values from the external data source. Objects in the OOHDB that are drawn from external data sources will have no attribute data stored within the OOHDB—all attribute values will

be "null". The approach shown in Figure 3.4b provides a uniform hierarchy for modeling "native" objects whose data are stored within the OOHDB as well as objects stored externally. The *DEO* and *DEOKey* attributes of native objects will be "null", indicating that the data is stored within the OOHDB.



(a) Typical Inheritance (b) Inheriting DEO Attributes

Figure 3.4: Inheriting From a New Root Class

The DEO classes form a separate hierarchy that is used to encapsulate the details of accessing external data sources. Figure 3.5 shows how one branch of the hierarchy supports inheritance among domain classes while the other branch is used to encapsulate the implementation details of accessing external data sources.

Our approach has the advantage that the class hierarchy containing domain schema class definitions remains free of external data source implementation details. New data sources can easily be added to the OOHDB by developing new DEOs without having to reorganize an already cluttered class hierarchy.

Each Schema Layer object in the OOHDB refers to the DEO that encapsulates the data source where the object's attribute data is located. The object stores a DEOKey providing enough information for the DEO to identify and retrieve the Schema Layer

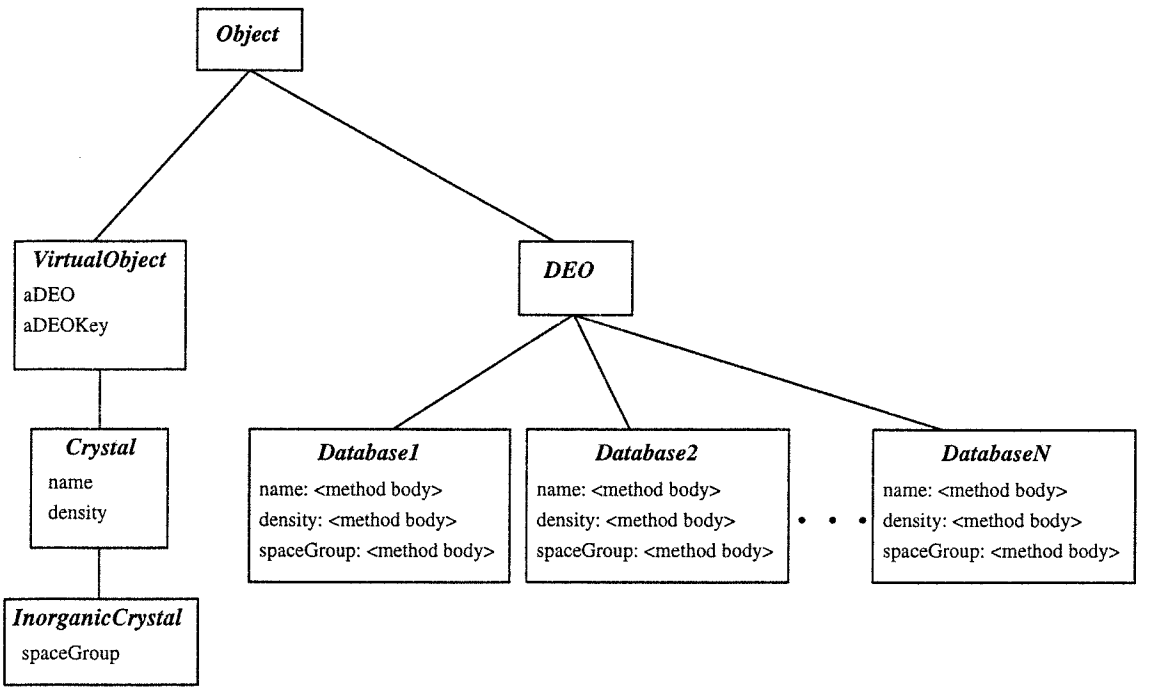


Figure 3.5: Separating Implementation from Inheritance

object's data when required.

When an attribute of a Schema Layer object is accessed via an accessing method, that method passes a request along to the DEO to retrieve the data. For example, a C++ method for accessing an attribute may do nothing more than return the attribute value:

Pseudo-Code Fragment 3.1 Basic Attribute Access

```
float density() {
    return this->density;
}
```

Our approach requires the use of attribute accessing methods that are extended to forward the message along to the DEO if the attribute value is "null", otherwise just return the value of the attribute:

Pseudo-Code Fragment 3.2 Attribute Access With DEOs

```
float density() {
    if (this->density == null)
        return this->DEO->density(this->DEOKey);
    else
        return this->density;
}
```

The format of the DEOKey attribute will depend solely on the method used by the DEO to access the external data source. For example, the DEOKey for an object stored in a relational database might be the set of *(attribute name, value)* pairs that comprise a primary key. An object stored in a file of formatted data records might use a DEOKey consisting of the starting byte position in the file and the number of bytes that make up the record. An object whose data is stored in a single file might simply use the file's pathname as a DEOKey.

The method forwarding approach can easily be extended to allow the OOHDB to be populated with a mixture of external and native objects. We extend Fragment 3.1 to test the value of the attribute *DEO* to determine whether this object is located within the OOHDB (DEO == null) or an external data source (DEO != null):

Pseudo-Code Fragment 3.3 Uniform Access to Internal and External Data

```
float density() {
    if (this->density == null) && (this->DEO != null)
        return this->DEO->density(this->DEOKey);
    else
        return this->density;
}
```

However, not all OODBMSs have a data model that *require* attributes to be accessed via methods. Some OODBMSs allow direct "structural access" to attribute values. In order to use our approach, attributes must be accessed primarily using methods. Requiring access by methods forces a programming discipline onto developers of the OOHDB so that they are careful to avoid structural access and rely on method-based access instead. In Section 3.3 we will discuss a few exceptions to this rule that are designed to enhance performance.

Forwarding messages to the DEO provides a simple and powerful mechanism for transparently retrieving data from external data sources on demand.

3.1.2 Database Encapsulation Object Layer

A DEO encapsulates all access to a single external data source and provides a homogeneous interface to all of the schema objects drawn from that data source.

A DEO has three components. First, there is the method interface used by Schema Layer objects to request data. Second, there is the interface to the external data source that is used to retrieve data. Third, between these two, there is the method code that maps the data from its heterogeneous external representation to the homogeneous representation of the schema. Thus, though not of critical importance, we think of the DEO as a three-layered entity itself as shown in Figure 3.6. Coincidentally, this three-layered design for the DEO corresponds to Gio Wiederhold's proposal for "mediators" that provide an interface between users and diverse data resources [Wie92]. The principle difference is that the "users" of a DEO are the objects of the user's schema rather than the users themselves.

The top layer of the DEO is the homogeneous method interface used by Schema Layer objects to request particular attributes to be retrieved. In Figure 3.6, the DEO provides an interface for the *name* and *density* methods.

In the middle layer of the DEO, data is mapped from its external representation to the representation used by the OOHDB schema. In Figure 3.6, for example, the *name* method simply extracts the string from the external data while the *density* method computes the density using the volume and atomic mass of the crystal's unit cell. These are trivial examples of the powerful computation and conversion that a DEO can perform in order to mask heterogeneity among data sources.

At the bottom of the DEO is the program code that communicates with an external data source to retrieve data. The example in Figure 3.6 retrieves a record from a relational database. The "where" clause of the query is constructed at runtime depending on the DEOKey of the object that requested the data.

Notice that the granularity of retrieval between the DEO and the external data source is intended to be "large-grained". In our example, the DEO queries the database to

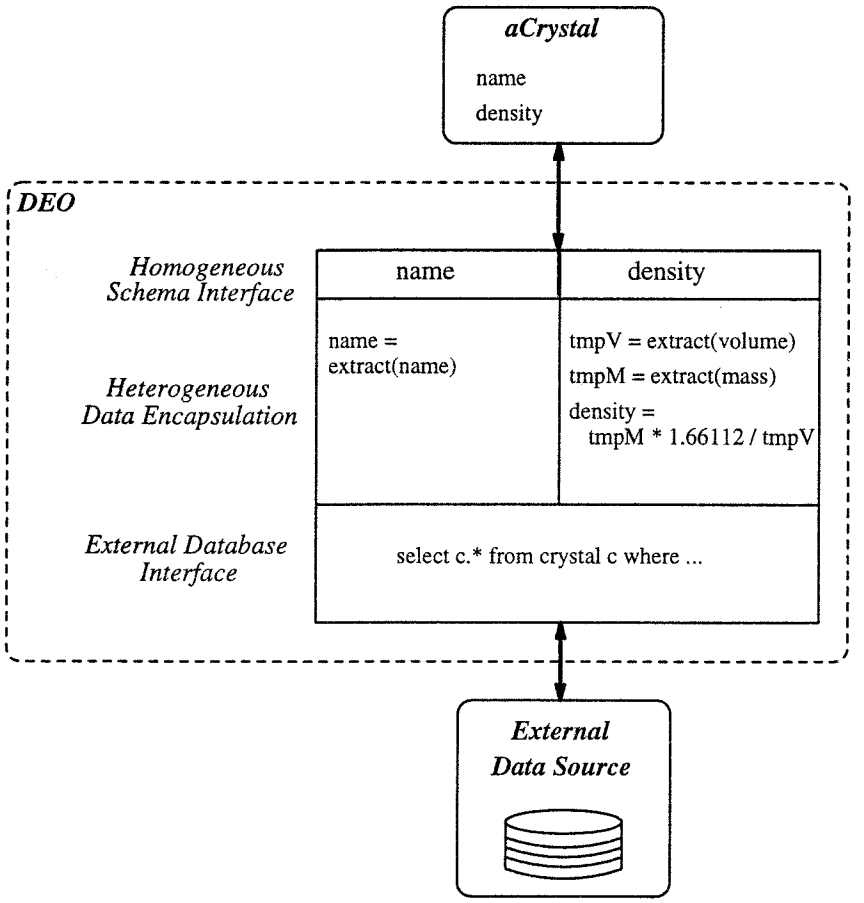


Figure 3.6: The Database Encapsulation Layer

retrieve *all* of the attributes of the record being accessed (e.g., select c.*) rather than just the name or density attribute. There are two reasons for this large-grained access. First, choosing a maximal retrieval scheme where all of an object's data is retrieved at once greatly simplified the DEO. This one-size-fits-all approach eliminates complex processing to build custom queries to extract only the required data. Second, accessing a record from an external data source is a time-consuming operation. By retrieving the extra attribute values and then caching the data temporarily, we eliminate most of the cost of accessing additional attributes from the same object. In Section 3.3 we discuss the benefits of caching data in more detail.

3.1.3 External Data Source Layer

Though we consider the external data sources a layer of our architecture, this layer is primarily comprised of the external sources and the software used to access their data. The only requirement imposed by our architecture on this layer is that the data be accessible. For data managed by a DBMS, we propose to use the API provided by the DBMS to access the data. For a lightly-managed database, the access may be as simple as extracting some portion of a file storing multiple entities, or retrieving an entire file that stores a single entity.

Where general-purpose access is not available for accessing lightly-managed data, we develop simple, generic interfaces to broad classes of data sources. For example, in Section 4.3.3 we describe two simple, generic BSD socket-based applications: one for retrieving a file given its pathname, and the second for retrieving a specified number of bytes from a specified location in a large file. These two socket-based "servers" are examples of the sort of minimal interface to a lightly-managed data source required by the OOHDB.

3.2 Objects, DEOs, and Databases—An Example Query

As an example of how the architecture processes queries, consider a simple OOHDB with the Schema Layer populated with *Crystal* objects drawn from external data sources. Assuming that the OOHDB has a collection in the Schema Layer containing all the *Crystal* objects called *AllCrystals*, the following C++-like query would be used to print the names of all the "dense" crystals:

Pseudo-Code Fragment 3.4 A Simple Query

```
Crystal *c;
foreach (c, AllCrystals)
    if (c->density() > 10.0)
        cout << c->name();
```

This query iterates over the *AllCrystals* collection, sending each *Crystal* the *density* message and then testing the result against the value 10.0.

Figure 3.7 demonstrates the processing performed by a particular *Crystal* in order to respond to the *density* message. In the course of a query over *AllCrystals*, each *Crystal*

object in turn receives a message asking for the value of its *density* attribute. If the *density* is "null", then the DEO (if it exists) is asked to retrieve the data from the external data source. The DEO first checks its cache, then optionally retrieves the entire record from the external data source. The atomic mass and volume values are extracted and used to compute the density value that is returned as the answer to the original message.

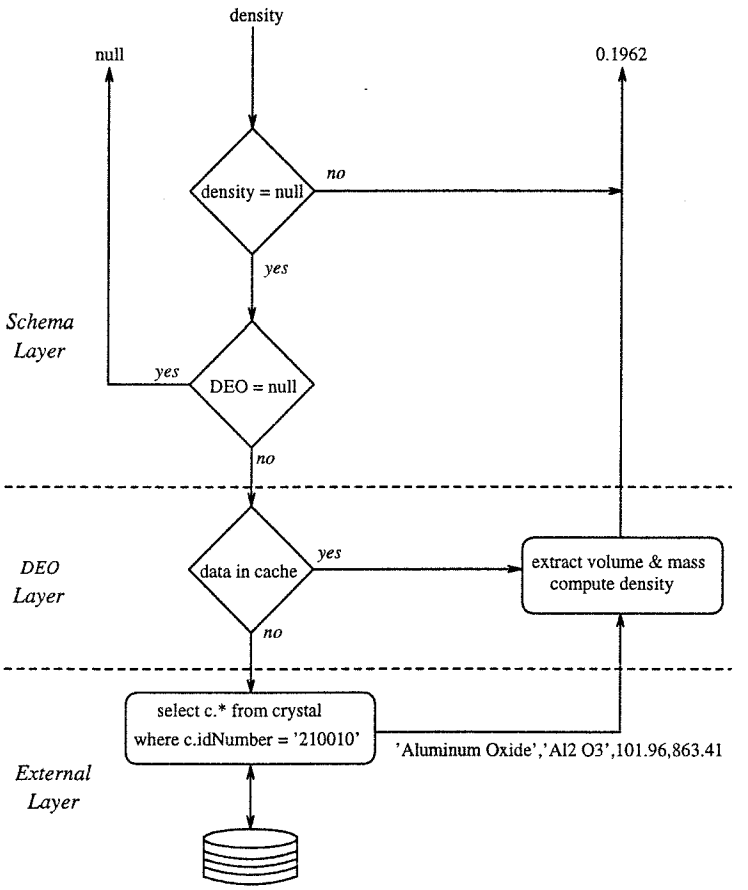


Figure 3.7: Message Execution Flowchart

This approach provides a high degree of transparency for users of the OOHDB. The details of accessing the external data source and homogenizing the data are completely hidden and automatic. However, the latency of accessing external data sources is *not* hidden from the users. In fact, as described so far, our approach is likely to be unusable for even moderately large collections of objects in the OOHDB. Fortunately, the OODBMS provides mechanisms that can be used to address this performance problem.

3.3 Making it Go Fast

The underlying OODBMS can be used to provide a number of performance-enhancing mechanisms for our OOHDB. The most important mechanisms are caching external data within the OOHDB and creating indexes. The ultimate goal when employing these tuning mechanisms is to achieve query performance that approaches the native, optimized performance of the underlying OODBMS.

3.3.1 Caching and Retaining Data

The easiest way to overcome the cost of accessing external data sources is to minimize the need to access them by caching or retaining data that is frequently used by queries. Of course, the obvious question is why bother retaining bits and pieces—why not simply load all external data into the OODBMS? The primary reasons are space and resource utilization. A part of the lure of any HDB design is the ability to provide access to very large databases. These large databases, taken together, may be larger than can reasonably be managed by a single database management system or hardware platform. Distributing a heterogeneous database across a number of systems often hides the magnitude of the whole database. Furthermore, users of large databases are often interested in only a small subset of the data at any particular moment. Dedicating large computational and storage resources to managing large amounts of rarely accessed data is not a wise use of resources.

Both DEO objects and Schema Layer objects provide reasonable locations for caching and retaining data.

As we noted in Section 3.1.2, having each DEO cache recently read raw data records eliminates wastefully re-reading the same record when more than one attribute is needed. So we extend the design of each DEO's methods to first check a local cache within the OOHDB. Our initial intuition was that a one-line cache containing the most recently read data would be sufficient. The iterative nature of OODBMS queries suggested that the cache would only be useful for accessing multiple attributes from the object referenced by the iterator. However, as we will see in Chapter 4, the large performance penalty of accessing external data suggests that a larger cache, one capable of holding a realistic

"working set" of data, provides a substantial performance improvement. We propose to manage the cache using a simple FIFO policy that minimizes the amount of bookkeeping required.

However, the cache cannot be an attribute of the shared DEO object as that could create a concurrency conflict among concurrent users. As data is read from external sources, the DEO cache is updated. Since the cache appears to the OODBMS' transaction mechanism as a shared resource concurrent users accessing and updating the same DEO cache would cause concurrency conflicts. Users would either be unable to concurrently access the same external data source (under pessimistic concurrency control) or their transactions would fail when the conflict was detected (under optimistic concurrency control). The solution is for each OOHDB user to have a private cache for each DEO. The DEO will need to identify which cache to use with a name that will be resolved at runtime. This duplication has two drawbacks however. First, two users accessing the same data will duplicate much of the raw data in each others DEO caches. Second, by dividing the available space in the OODBMS among multiple caches, the size of each cache is reduced.

In order to support caching raw data, an early prototype of our architecture naively created a private DEO object for *every* Schema Layer object [HMSW92]. The reasoning was that these individual DEOs would provide a convenient location for caching raw data read from external data sources. We hoped to be able to manage this caching to provide fast access for small sets of objects that were being frequently accessed. However, this strategy proved ineffective (due to the iterative nature of queries noted earlier) and consumed large amounts of space within the OOHDB—space that was difficult to reclaim. A single DEO with small private user caches has proven to be a more reasonable and effective solution.

Likewise, Schema Layer objects can retain[1] the attribute values retrieved from external data sources. Attribute values can be retained as they pass through the Schema Layer objects by simply storing the value in the object's attribute. A simple extension to

[1]We differentiate between "caching" and "retaining" data. A cache is a shared fixed-sized data space that may hold any data. On the other hand, an attribute of a particular object is not shared and has only one value. Thus we talk of caching raw data but retaining attribute values.

Fragment 3.3 supports retaining attributes:

Pseudo-Code Fragment 3.5 Attribute Access With Retention

```
float density() {
    if (this->density == null && this->DEO != null)
        if (retainAttributes)
        {
            this->density = this->DEO->density(this->DEOKey);
            return this->density;
        }
        else
            return this->DEO->density(this->DEOKey);
    else
        return this->density;
}
```

Here we assume some flag, "retainAttributes", that is set "true" or "false" by the OOHDB user.

Retaining attribute values has two great benefits. First, the entire process of reading and converting data from the external data source is eliminated—attribute data is accessed using the native mechanisms of the OODBMS, all OOHDB extensions are bypassed. Second, and most importantly, indexes can be constructed using these attribute values to provide optimized query performance.

3.3.2 Indexing

By indexing large collections in the OOHDB along commonly accessed paths, many routine queries over these collections are processed without having to access external data sources. Constructing an index trades space in the OOHDB for increased query performance. Although all indexes consume space, our architecture carries the additional cost of allocating space for the retained attribute values.

While some external data sources may already provide indexed access, their indexes are not available to the query engine of the OODBMS and cannot be used. Furthermore, the indexes supplied with external data sources are pre-determined and will vary from source to source. However, using the indexing mechanism of the OODBMS, the OOHDB provides uniformly optimized access across all external data sources. Furthermore, new optimized

access paths can be added to the OOHDB to suit the needs of its users. Thus, while the OOHDB may not be able to take advantage of an external data source's optimizations, it is not hampered by the lack of such optimizations either. Since we expect lightly-managed databases to be light on optimized access mechanisms, we believe that our architecture presents a reasonable approach to optimization.

In order to construct an index then, the "retainAttributes" flag is turned on, the attributes participating in the index are accessed (and thus retained), and the index constructed using OODBMS indexing mechanisms:

Pseudo-Code Fragment 3.6 Building an Index

```
Crystal *c;
retainAttributes = TRUE;
foreach (c, AllCrystals)
    c->density();
AllStudents->createIndex('density');
retainAttributes = FALSE;
```

One problem with retaining attribute values is that it is difficult to manage and reclaim space from retained attributes. Management is not as easy as the simple FIFO management of a DEO cache. It is difficult, if not impossible to automatically determine which attribute values should no longer be retained. Thus our architecture does not provide a management strategy or mechanism for disposing of retained data. The users of the OOHDB can most accurately determine when attribute values should have their space reclaimed by explicit disposal or by severing reachability (i.e., resetting the attribute value to "null") and collecting garbage.

3.4 Populating and Maintaining an OOHDB

One detail remains—how an OOHDB is populated and maintained. Populating the OOHDB requires the OOHDB to be able to identify each entity in an external data source that will be represented by an object in the OOHDB. Entities stored in a relational database, for example, can be identified with a simple query to retrieve all primary keys (e.g., *select c.id from c in Crystals*).

On the other hand, lightly-managed data sources do not provide such a convenient query interface. We expect that data files storing single entities (e.g., scientific data-interchange file formats) are likely to be managed using some sort of directory structure. In such an environment, scanning the directory to obtain the file pathname is sufficient for populating the OOHDB. New Schema Layer objects are created for each file in the directory and the pathname of the file is used as the DEOKey:

Pseudo-Code Fragment 3.7 Scanning a Directory

```
struct direct **filenames;
Crystal c;
scandir("/data/netCDF/", &filenames, NULL, NULL);
for(i=0; i<sizeof(*filenames); i++)
{
    c = Crystal->new;
    c->DEO = netCDFFile;
    c->DEOKey = (*filenames)[i]->name;
    AllCrystals->add(c);
}
```

Small numbers of single files may simply be added to the OOHDB manually by creating an object and providing the file pathname as the DEOKey.

The other sort of lightly-managed data source we encounter are the large data files that contain many entities. This sort of database is highly structured and the structure can be exploited to identify the records (or entities) in the file. A Record typically encodes information that identifies its beginning and possibly its end. These identifying bits of syntax can be used in a linear scan of the file to populate the OOHDB with an object for each entity identified. For example, a file using 80-column FORTRAN-like "cards" might identify the first card of each record with a special character in column 80. We expect this sort of file to be accessible via a general-purpose interface for reading from the file. Fragment 3.8 is an example of the sort of processing required for scanning a file that is accessible via a BSD socket-based server responding to requests to read a given number of bytes from the file. The code simply steps through the file in 80-byte pieces creating new Schema Layer objects each time the beginning of a record is detected (denoted by a

special character in byte 80 in this example):

Pseudo-Code Fragment 3.8 Scanning a Multi-Record File

```
socket nbsDatabaseServer;
char[80] inbuf;
char[80] outbuf;
int bytesRead, position = 0;
Crystal c;

/* Request 80 bytes starting at the specified position */
scanf(outbuf, "80 %d", position);
write(nbsDatabaseServer, outbuf, sizeof(outbuf));
bytesRead = read(nbsDatabaseServer, inbuf, sizeof(inbuf));
/* Repeat until there we reach the end of the file */
while (bytesRead == 80) do
{
    /* Is this the beginning of a new record? */
    if (inbuf[79] = '*')
    {
        /* Patch the last record's end position */
        if (c != NULL)
            c->DEOKey[1] = position-1;
        /* Create a new Crystal and initialize the DEO and key */
        c = Crystal->new;
        c->DEO = NBSCrystalDatabase;
        c->DEOKey = NBSKey;  /* a 2-integer array */
        c->DEOKey[0] = position;  /* byte offset where the record begins */
        AllCrystals->add(c);
    }
    /* Increment the position in the file and request the next 80 bytes */
    position += 80;
    scanf(outbuf, "80 %d", position);
    write(nbsDatabaseServer, outbuf, sizeof(outbuf));
    bytesRead = read(nbsDatabaseServer, inbuf, sizeof(inbuf));
}
```

Since the population and maintenance functions are dependent on the structure of each external data source, it makes sense to encapsulate these operations within the DEO. The initial population operation creates Schema Layer objects within the OOHDB for each entity identified in the external data source. Subsequent update operations will depend largely on the way the external data source is managed. At one extreme are the external

data sources that are read-only repositories of legacy information. In this case, an update operation may not even be necessary. On the other extreme are the frequently updated data sources where anything is liable to change. In this case deletions, insertions, and updates to the external data must all be detected and reflected in the OOHDB. For very active data sources, detecting modifications may be a non-trivial effort and our approach may be unsuitable.

3.5 Summary

In summary, we wish to evaluate the design of our architecture in terms of the two goals we presented at the beginning of this chapter as well as Won Kim's 9 Objectives for an HDB from Chapter 1.

3.5.1 Did We Achieve Our Goals?

We believe this OOHDB is easy to implement using a wide variety of COTS OODBMS products. We have used this architecture to implement an OOHDB for materials science using the GemStone OODBMS (see Chapter 4). The architecture and methodology for constructing an OOHDB outlined here relies on the basic features common to OODBMSs—an object-oriented data model that encapsulates object state and behavior, supplemented with a BSD socket interface to external software components. The minimal functionality required of an OODBMS by our architecture is examined further in Chapter 5 where we examine other COTS OODBMSs as tools for implementing an OOHDB based upon our architecture.

We believe that the OOHDB is uniquely suited for integrating lightly-managed databases. The key to integrating these databases is to build a representation of their data within the OOHDB. Rather than using the class hierarchy to hide heterogeneity, we use a mechanism based on forwarding messages. The approach involves extending each Schema Layer class's attribute accessing methods to forward data access requests on to a DEO. These DEOs provide a homogeneous interface to heterogeneous data using methods that map external data to the homogeneous representation of the OOHDB.

Our multi-layered architecture leverages the power of the underlying OODBMS to provide optimized query processing over heterogeneous data stored in external data sources.

3.5.2 Do We Meet Kim's Objectives

In Section 1.2 we presented Won Kim's objectives describing the capabilities of an HDB. An obvious question is whether our OOHDB meets these objectives.

OBJECTIVE 1 *obviate the need for batch conversion*—Data from external data sources is retrieved on demand by the OOHDB. The only batch-processing is the initial population and subsequent updates of the OOHDB.

OBJECTIVE 2 *no changes to local database system software*—The OOHDB uses the data access mechanisms provided by the external data source. For lightly-managed databases, simple non-intrusive, general-purpose data access "servers" access the data.

OBJECTIVE 3 *users of local databases are unaffected*—By using the data access mechanisms provided by the external data source, the OOHDB appears to the external data source as just another user. External data source users and their applications are unaffected.

OBJECTIVE 4 *single database language*—The OOHDB uses the data manipulation language of the underlying OODBMS. Users are completely shielded from the heterogeneous interfaces of the external data sources by the DEOs.

OBJECTIVE 5 *shield users from the heterogeneous operating environment*—A single, uniform data access mechanism is provided by the OOHDB that shields users from the heterogeneity of computer and operating systems, networks, DBMSs, etc.

OBJECTIVE 6 *distributed read/update transactions*—As presented here, the OOHDB does not provide distributed transaction management or updates. However, users accessing the OOHDB do so using the transaction mechanisms of the underlying OODBMS. We believe that, where necessary and feasible, the transaction environment of the OODBMS could be extended to coordinate distributed transactions

among external data sources that support distributed transactions. Modifications to attribute updating methods could be used to provide distributed updates to the external data sources from within a transaction environment. Of course, distributed transactions and updates are still a significant research issue. In Chapter 6 we discuss supporting distributed transactions and updates from the OOHDB as a subject for future research.

OBJECTIVE 7 *must be full-blown database system*—Clearly, the underlying OODBMS *is* a full-blown database system. By constructing our OOHDB using an OODBMS we leverage the schema definition, query processing and optimization, transaction management, concurrency control, etc. of the OODBMS. The OODBMS will also typically provide a host-language API for user programs to access the OOHDB as well as interactive query facilities. Again, the ability to update data via the OOHDB remains a research issue.

OBJECTIVE 8 *no changes to local database administration*—No changes to external data source administration are required since the OOHDB interacts with those data sources as an ordinary client. However, it may be the case that changes to external data sources might be used to help maintain the OOHDB. For example, an external database might be enhanced to write a change log that the OOHDB could use to maintain its representation of the external database more efficiently.

OBJECTIVE 9 *performance approaching homogeneous distributed system*—Strategies for enhancing performance, such as caching raw data and retaining attribute values, should be able to provide performance that approaches a homogeneous *non*-distributed system. Where queries can be satisfied using retained attributes and indexes within the OOHDB, performance will be equivalent to the native performance of the underlying OODBMS. Furthermore, our approach gives the users of the OOHDB control over query optimization. Users are insulated from changes to external data sources that may affect performance (e.g., the removal of an index). New optimizations can be added to the OOHDB using the mechanisms provided by the OODBMS (e.g., indexes). The performance aspects of our architecture are

further explored in Chapter 4 where we measure the performance of an OOHDB for materials science.

Clearly our architecture meets most of Kim's criteria for an HDB. Leveraging the power of an OODBMS for constructing an OOHDB allows us to meet many of these criteria with little or no additional effort. Most importantly, we are able to meet these criteria and provide high-level, optimized query performance for lightly-managed databases.

In Chapter 4, we discuss our efforts to take this high-level, OODBMS-independent architecture and apply it to implementing an OOHDB for a scientific domain, materials science.

Chapter 4

An Implementation of the OOHDB Architecture

We have taken our architecture and implemented an OOHDB for materials scientists and lightly-managed materials science data. The implementation of a materials science OOHDB serves two primary purposes. First, it demonstrates that our architecture can be constructed using commercial OODBMS technology. Second, it provides a testbed for examining and analyzing the performance of our OOHDB architecture in a non-trivial, real-world application.

4.1 The OODBMS—GemStone

GemStone, from Servio Corporation[1], is a commercial object-oriented database management system that evolved from object-oriented database research conducted in the mid-80's [CM84, MS90, BOS91].

GemStone uses a client-server architecture with a single server ("stone") and multiple clients ("gems")[2].

A database is logically broken into "segments", each user typically owning and controlling a segment. Persistence is by reachability with both manual and scheduled automatic garbage-collection of non-reachable data. GemStone supports the creation of indexes over non-sequenceable collections (e.g., sets and bags) by specifying a "path"

[1] Servio Corporation, 15400 N.W. Greenbrier Parkway, Suite 280, Beaverton, Oregon 97006.

[2] GemStone further allows the "gem" portion of the client to be separated from the application, providing for an application-client-server distribution of processes.

(e.g., molecule.formula.atoms.element.symbol). An index can be based on the immutable object-identifiers (OID) of the objects in the collection, or on the equality of attribute values.

The data model of GemStone is based on the language Smalltalk-80 [GR83, KP86] and provides a rich class-hierarchy of pre-defined classes (e.g., set, bag, array, integer, float, boolean, string, dictionary, queue). The data definition and data manipulation language is called "SmalltalkDB" and is a superset of the Smalltalk-80 language. Methods are coded using SmalltalkDB and are compiled[3] and stored within the database.

GemStone provides interactive user interfaces as well as application programmer interfaces for C and C++ and a Smalltalk interface compatible with Objectworks\Smalltalk from ParcPlace Systems[4] and Smalltalk/V from Digitalk[5]. In addition, GemStone provides a fully self-contained application development environment called GeODE (GemStone Object Development Environment).

4.2 The Domain—Materials Science

We have chosen to use materials science as a test domain for our OOHDB implementation. Materials scientists have been leaders in the use of computers for modeling and research. Computational models for materials science are well known and refined [Hal85]. In addition, there are many computer-readable lightly-managed data sources available for materials science [HS91, Wil85, Ber85, Rum89, Mes84].

The particular problem we sought to address was to provide an integrated OOHDB for materials scientists and their application programs. The OOHDB was to initially integrate diverse sources of crystallographic data and be easily extended to integrate other related materials science sub-domains, such as phase-diagram calculation, to provide a wide-ranging information resource.

[3]Smalltalk methods are "compiled" into a machine-independent intermediate code that is interpreted by the Smalltalk virtual machine at runtime.

[4]ParcPlace Systems, 1550 Plymouth Street, Mountain View, California 94043.

[5]Digitalk Inc., 9841 Airport Blvd., Los Angeles, California 90045

Previously, applications such as the Desktop Microscopist—an application for generating computer-predicted diffraction patterns—accessed lightly-managed data sources directly. However, as new data sources were added, the application's data model and input-output routines were forced to change and adapt to the new data source. Furthermore, application users did not have access to a general query facility and were restricted to accessing the data using a small number of pre-selected queries. The OOHDB provides an application-independent interface and powerful query language for users and their applications and can integrate a wide range of materials science data.

4.2.1 Materials Science Schema

We began by developing an application-independent object-oriented model for materials science data. As shown in Figure 4.1, the data model of the Schema Layer currently supports a sub-domain of materials science, known as crystallography or structural analysis. Obviously, developing a uniform data model for the domain is a key ingredient for building an OOHDB. In developing our model we have attempted to take a broad view of materials science as a whole rather than limiting the model to the sub-domain of crystallography. Thus, while crystallographers might reasonably model data with the "crystal" at the heart of the model, we model materials as complex objects of sub-components (e.g., *atoms* comprise *molecules* comprise *crystals* comprise...). Our intention is to provide a robust data model that can be extended to other sub-domains of materials science, such as phase-diagram calculation.

Briefly, the important classes in Figure 4.1 include:

Element — the basic information for a periodic table element (e.g., atomic mass, atomic number, sets of valid nuclear and electron configurations).

Element Configuration — a configuration of the electron shell for a particular element (i.e., the number of electrons and their locations)

Nuclear Configuration — a configuration of the nucleus for a particular element (e.g., the number of neutrons in the nucleus).

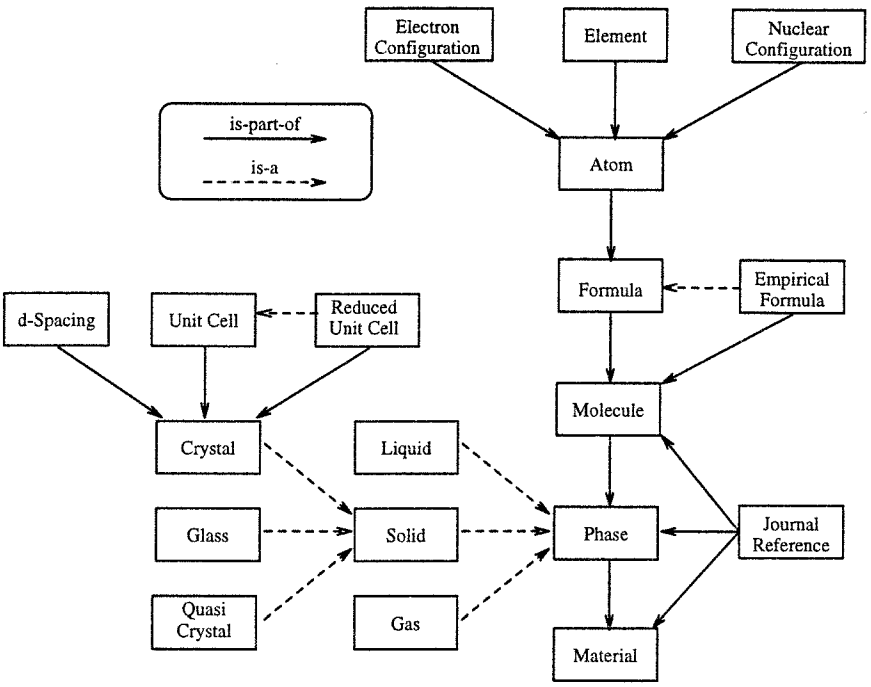


Figure 4.1: Object-Oriented Data Model for Materials Science

Atom — an element with a particular electron and nuclear configuration.

Formula — a list of atoms and coefficients describing a chemical formula (e.g., H_2O).

Molecule — a particular chemical and empirical formula along with other characteristics of the molecule (e.g., molecular weight, melting temperature).

Crystal — a solid phase of a particular molecule having a crystalline structure.

Unit Cell — a basic description of the crystalline structure in the form of the distances and angles that define the unit cell shape and size.

d-Spacing — measurements of the spacing and intensity of a diffraction pattern image generated by scanning the crystal with an electron microscope.

Material — a collection of molecules in various phases.

Journal Reference — identifies the journal where the data for the particular object was reported.

Our OOHDB focuses on integrating data for crystals and their unit cell data. Many of the other classes in Figure 4.1 simply provide basic structure for the model and are intended to support extending the model to other materials science sub-domains. Appendix A provides a description of the GemStone definitions for the classes of the Schema Layer as depicted in Figure 4.1.

4.2.2 Data Sources Integrated

Our materials science OOHDB provides access to three types of data sources:

1. Large commercial databases on CD-ROM.
2. Single files written by scientific application programs.
3. Single files written using the Crystallographic Interchange Format.

The large, commercial databases are the National Bureau of Standards *Crystal* database, and the International Centre for Diffraction Data *PDF-2* database. Both are available from the National Institutes of Standards and Technology (NIST) distributed on CD-ROM [NIS92]. Both databases share a common format, NBS*AIDS83 [Int90]—an ASCII file-format using a variable number of 80-column "cards" for each crystal record (see Appendix B.1 for an example). There are 22 card types defined by the NBS*AIDS83 document. The format of each card type is fixed (i.e., the definition of the card determines the position and format of data). Accessing attributes is thus a matter of extracting a fixed number of bytes from a fixed position in a card. Some card types may be repeated when necessary (e.g., multiple authors of a journal article documenting the crystal structure). Some card types are required (e.g., card "1"—unit cell parameters), others are optional (e.g., card "B"—comments). The Crystal database contains data on approximately 136,000 crystals and occupies 167 megabytes. The PDF-2 database contains data on approximately 51,000 crystals and occupies 127 megabytes[6], with record sizes averaging 1K–3K bytes.

[6]PDF-2 records are typically larger than Crystal records because they contain a number of optional cards (most notably for d-spacings) that are not contained in the Crystal records.

The OOHDB also provides access to data files that have been written by application programs. Both the Desktop Microscopist[7] and the CAChe[8] computer-aided chemistry program can store data in single ASCII files (one molecule or crystal per file). The files written by the CAChe system and the Desktop Microscopist use different formats, but are similarly comprised of (keyword,value) pairs such as "Space Group = 193" (see Appendix B.2 and B.3 for examples). Thus, the format of these files are self-describing (though not sufficiently to be read by humans) and not as rigid as the format of the NBS databases. Attributes must be located by using a string search to find the keyword ("Space Group") and then extract the associated value ("193"). Desktop Microscopist files are typically 1K–3K bytes. CAChe system files can be much larger as they are intended to support larger molecules and a wider range of applications. Typical file sizes range from 2K–40K bytes.

Finally, the OOHDB provides access to files written using the Crystallographic Interchange Format (CIF), based on the STAR self-describing file format [HAB91]. CIF is one of an emerging number of "data interchange format" standards designed to promote machine and program-independent file formats for exchanging data. CIF files are self-describing ASCII files that can be viewed as structured groups of (keyword,value) pairs (see Appendix B.4 for an example). The CIF format is extremely flexible. However, a data dictionary defines the basic elements and their formats for particular domains. CIF syntax has just a few structural elements including "data" segments, "loops", and "elements". Basically, a data segment is a sequence of elements, each segment typically comprising a single "record" with the potential for multiple records per CIF. Each element is a (keyword,value) pair. What differentiates CIF files from other file formats is that CIF uses a context-sensitive syntax of (keyword,value) elements. That is, a CIF allows the "value" to be a structured loop of other (keyword,value) elements[9]. Thus the simple string searching used to locate attributes and their values for Desktop Microscopist and CAChe files is not sufficient. A CIF must be parsed in order to determine whether an element is a part of

[7]The Desktop Microscopist is distributed by Virtual Laboratories, Ukiah, California.
[8]CAChe is a registered trademark of CAChe Scientific.
[9]CIF does not allow loops within loops, although the underlying STAR format allows such nesting.

a larger structure (i.e., a loop) before the data can be extracted. File sizes are extremely variable. The verbose nature of CIF files is designed to foster human-readability and tends to make them large (the test file we used was approximately 30K bytes for a single crystal).

4.3 Our Layered Implementation

The three layers of the design (Section 3.1) were easily mapped into a GemStone implementation. OOHDB users access data using the materials science schema of Figure 4.1. DEOs were created for each type of external data source accessed by the OOHDB and two simple, general-purpose, platform and data-independent "servers" provide access to external data sources.

4.3.1 The Schema Layer

Each of the materials science classes of Figure 4.1 is rooted in the class *VirtualObject*—a subclass of the GemStone root class *Object*. *VirtualObject* adds two attributes to each subclass—*deo* and *deoKey*.

As described in Sections 3.1.1 and 3.3.2, we extend the behavior of each attribute accessing method to forward messages on to the DEO when we need to retrieve data from an external data source. In GemStone, an object's attributes are typically accessed via "accessing" methods bearing the same name as the attribute accessed. Normally, an attribute accessing method simply returns the value of the attribute[10]:

SmalltalkDB Fragment 4.1 Basic "spaceGroup" Accessing Method

```
spaceGroup
    "Return the value of the instance variable 'spaceGroup'."
    ^spaceGroup
```

[10]Caret (^) is equivalent to "return" and can be applied to variables or an expression where the result of evaluating the expression is returned.

In programming the OOHDB we extend these accessing methods to check for the presence of a DEO, forward messages to the DEO if the attribute value is "nil", and potentially retain the attribute value:

SmalltalkDB Fragment 4.2 OOHDB "spaceGroup" Accessing Method

```
spaceGroup
    "Return the value of the instance variable 'spaceGroup'."
    ( (spaceGroup isNil) and: [deo notNil] ) ifTrue: [
        virtualObjectCaching ifTrue: [
            ^spaceGroup := deo spaceGroup: deoKey]
        ifFalse: [^deo spaceGroup: deoKey].
    ifFalse: [^spaceGroup]
```

These six lines of code are an example of the conceptual template used by all attribute accessing methods.

Our approach to integrating data has influenced how we manage the objects of the schema. One of our objectives is to use as little space as possible within the OOHDB to represent external data. However, when we create a new Schema Layer object (e.g., a *Crystal*), GemStone pre-allocates space for each attribute of the object. A single *Crystal* object uses 55 bytes of storage in our GemStone OOHDB[11]. Except for the *deo* and *deoKey* attributes, all other attributes initially point to the undefined "nil" object. When we created *Crystal* objects in the early versions of our materials science OOHDB, we eagerly instantiated all of the related sub-objects as well—*JournalReference*, *UnitCell*, *ReducedUnitCell*, etc. What we observed was that since these sub-objects are not shared among *Crystals*, we could save considerable space in the OOHDB if we delayed the creation of these sub-objects. Thus, we allow the *unitCell*, *reducedUnitCell*, and *journalReference* attributes of a *Crystal* object to remain "nil" rather than pointing to a new, empty object of the appropriate class. When reference is made to a sub-object using an accessing method, a new object is dynamically created by the DEO and returned using the same mechanism demonstrated above for the *spaceGroup* method.

[11] GemStone does not provide a mechanism for directly assessing the space used by an object. All space measurements presented here have been made by checking the overall free space in the database, generating thousands of new objects, then re-checking the space available to determine how much space the new objects used.

While the space saved by delaying the creation of a particular sub-object is small (a *UnitCell* has only 7 16-bit floats, for example), the cumulative effect for hundreds of thousands of objects is substantial. Our materials science OOHDB currently contains approximately 206,000 *Crystals* consuming approximately 11 megabytes of space. Assuming that a *Crystal*'s sub-objects would each occupy approximately 55 bytes of storage as well, we estimate that delayed creation of *Crystal* sub-objects saves approximately $55 * 3 * 206,000 = 34$ megabytes. While 34 megabytes is a large amount of space by today's standards, shrinking the space required by a factor of 3 will become significant as the OOHDB grows. Of course there is some overhead imposed by dynamically creating objects on demand. However, the overhead appears to be negligible at about 0.9 milliseconds per object.

4.3.2 Database Encapsulation Object Layer

A DEO class[12] with methods for data retrieval and conversion is implemented for each external data source. Appendix C provides a list of the DEO classes, their class attributes and class methods. Each DEO is constructed similarly, and where possible, we have chosen to model common characteristics using a small class hierarchy of DEOs. For example, the *NBSDatabaseRecord* class encapsulates the common aspects of the nearly identical *CrystalRecord* and *PDFRecord* DEOs. Similarly, the common file access mechanism of the *DMFile*, *CACheFile*, and *CIF* DEOs are encapsulated in their common superclass *DatabaseFile*.

In Section 3.1.2 we characterized the DEO as a component with three layers. At the "top" of the DEO is the interface to the Schema Layer objects. The interface provides complete coverage of all messages the DEO might receive from Schema Layer objects. These interface methods use the general-purpose data extraction and conversion methods of the DEO to retrieve a particular attribute or object. The *spaceGroup* method of the PDF-2 database DEO provides an example:

[12] GemStone follows the Smalltalk model by allowing class definitions to contain both class attributes and class methods. When only a single instance of a particular class is to be instantiated, the class object itself can serve as both the definition *and* the only instance of the class.

SmalltalkDB Fragment 4.3 PDF-2 DEO "spaceGroup" Method

```
spaceGroup: aKey
    ^(self getFrom: aKey at: '3 ' from: 1 to: 8 dataType: #String)
```

This method uses the generic *getFrom:at:from:to:dataType:* method in the "middle" of the DEO to extract and convert the space-group value from the overall record. Interface methods for CIF, Desktop Microscopist and CAChe files are similar.

The "middle" of the DEO contains general-purpose methods for extracting a particular attribute from the entity and performing simple data-type conversions. The *getFrom:at:from:to:dataType:* method of the PDF-2 database DEO is a good example of the processing that takes place. The *getFrom:at:from:to:dataType:* method extracts and converts an attribute value from a PDF-2 record. It first checks the PDF-2 DEO cache to see if the record is already present. Once the record is in the cache, the record is sequentially scanned to locate the particular "card" that contains the data of interest (selected by the "aNumber" parameter). Each card is 80 characters long and the card number is located in byte 80. So the search for the proper card is accomplished by checking each 80th byte until the card is found or the end of the record is reached (in which case "nil" is returned to indicate missing data). If the proper card is found, the data in the range (from: to:) is extracted and converted to the proper data type. A special case affects "codes" from the NBS databases—short encodings of standard values designed to save space in the database. The PDF-2 and Crystal database DEOs include a "codeDictionary" class variable based on the GemStone *Dictionary* class that is used to map short codes to standardized strings.

SmalltalkDB Fragment 4.4 PDF-2 DEO "getFrom" Method

```
getFrom: aKey at: aNumber from: start to: end dataType: aDataType
    "Generic method to get data from the data record and convert to proper datatype"
    | tempString temp c t i recSize |

    "If the data isn't in the cache then get the data from the external file"
    temp := (self cache at: aKey ifAbsent: [self getData: aKey]).

    recSize := (aKey at: 2).
    i := 0.
    "Find the card that contains the data we're interested in.
```

```
If it exists then copy the range from the data, otherwise just return nil.
Check the 80th character of each card to locate the card containing the data."
[(tempString isNil) and: [((temp size) >= (i + 80)) and: [i < recSize ] ] ]
whileTrue: [
    (((temp at: (i + 80)) = (aNumber at: 1)) and:
            [('1G*+' includesValue:
   (temp at: (i + 80))) | ((temp at: (i + 71)) = (aNumber at: 2))])
    ifTrue: [tempString := temp copyFrom: (start + i) to: (end + i)].
    i := i + 80.
].
(tempString isNil) ifTrue: [^nil].

"Trim whitespace if this isn't a code"
(aDataType ~= #Code) ifTrue: [tempString := tempString trimWhiteSpaceIfNeeded].
"If in the course of truncating the data above it became empty then return nil"
(tempString = '') ifTrue: [^nil].

"Data is not nil so convert it to the datatype requested and return it."
"String?"
(aDataType = #String) ifTrue: [^tempString].
"Float?"
(aDataType = #Float) ifTrue: [^(SmallFloat fromString: tempString)].
"Integer?"
(aDataType = #Integer) ifTrue: [^(Integer fromString: tempString)].
"Code?"
(aDataType = #Code) ifTrue:
    [^(self codeDictionary at:
        ( ((aNumber copyFrom: 1 to: 1) add: (start asString))
            add: tempString) ifAbsent: [tempString]) ].
```

This method checks the DEO cache and uses *getData:* to retrieve the data from the external data source if necessary. The attribute is extracted, converted and returned. Some DEO objects have additional specialized methods for retrieving and building more complex types of objects (e.g., the *getdSpacings:onlyMax:* method of the PDF-2 database DEO that returns an Array of *dSpacing* objects).

The mechanism for extracting and converting attributes differs among the DEOs. In the example above, the data is located using a byte offset within a particular record. For the Desktop Microscopist and CAChe DEOs, the mechanism used to locate the data is to sequentially search for a label string within the file (e.g., "Space Group = 193" in the

Desktop Microscopist file in Appendix B.2).

Extracting data from CIF files is more complex. Though attributes are identified by a label within a CIF file, a sequential substring search is insufficient for locating and extracting an attribute. The problem is that an attribute may occur within the context of a larger data structure such as a loop. Thus a CIF file must be parsed to accurately locate and extract attributes. Our strategy is to convert the string representation of a CIF file to a structured representation based on the GemStone *Dictionary* class that stores (key,value) pairs. Thus a CIF data segment will be represented as a *CIFDictionary* where the keys are the data labels, and the values are either *CIFStrings* or *CIFLoops*—a subclass of *CIFDictionary*. The *getFrom:at:in:dataType:* method searches each *CIFDictionary* for an attribute with the given label. If not found, the message performs an in-order traversal of all embedded *CIFLoop* objects within the *CIFDictionary*.

One important lesson we have learned is that the format of an external data source can have a great impact on the performance of the OOHDB. Extracting attributes from the data of an external data source requires some modest computation. In general, the time needed to extract attributes seems to be a function of how rigidly the data is structured. In the case of the rigidly-structured NBS databases, the access mechanism retrieves a record from the CD-ROM by directly accessing the data based on a byte-offset into the file. The card containing the attribute is located by skipping through the record 80 bytes at a time until the card is found. The attribute is then extracted directly from the card based on the byte offset and length within the card. The more flexible Desktop Microscopist and CAChe file formats have higher complexity since they must be searched sequentially using substring matching. Searching a CIF file is even more complex since the file must be parsed into a structured representation, then searched. Although CIF files are not intended for data storage (as we are using them here) but for data interchange, data interchange formats are increasingly being used for data storage.

At the "bottom" of each DEO lies a *getData:* method that retrieves the data for a single entity from the external data source. Although all *getData:* methods share a common semantic meaning, the implementation of the *getData:* method for each DEO is dependent on the format and access mechanism of the external data source. The *getData:*

method for the PDF-2 database DEO is representative of the socket-based mechanism we use to retrieve a contiguous set of bytes from a large CD-ROM database. This method first gets a handle on the line in the DEO cache that is going to be replaced and the size of the cache line is increased if necessary to hold the new data. A read request is sent to the byte-server in the form of start and end byte positions of the data in the file. The data is read from the socket into the cache, the cache key is updated, and the data is returned.

SmalltalkDB Fragment 4.5 PDF-2 DEO "getData" Method

```
getData: aKey
    |sock c k s l|
    "Get the data from the external file server and place it in the cache."
    sock := self socket.
    self cache decrementLastIndex.
    l := self cache lineToReplace.
    k := l at: 1. "The cache Key"
    c := l at: 2. "The cache Data"
    "Make sure this line is large enough to hold the data - but we only want to
    grow, never shrink the lines"
    (c size) < (aKey at: 2) ifTrue: [c size: (aKey at: 2)].
    s := String withAll: 'r '.
    "Write the request with start/end positions.  Watch out for stale sockets!"
    (sock write:
       (((s add: ((aKey at: 1) asString)) add: ' ') add:
                ((aKey at: 2) asString))) <= 0 ifTrue: [^nil].
    "Read the data from the socket"
    sock read: (aKey at: 2) into: c.
    "Update the value of the cache key"
    k at: 1 put: (aKey at: 1).
    k at: 2 put: (aKey at: 2).
    ^c.
```

Notice that we are very careful to reuse the space in previously allocated cache lines by overwriting old data using the *read:into:* method. Reusing space is critical as it prevents old cache data from simply becoming garbage that must be collected before the space can be reused. If we naively allocated a new cache line each time we accessed an external data source, the GemStone OODBMS would rapidly fill with the unreachable garbage of previous cache lines and would periodically slow to collect all that garbage once space was exhausted.

For the file-based CIF, Desktop Microscopist, and CAChe data sources, the *getData:* method is similar, but the key is a combination of host and file pathname.

In addition, each DEO also provides a method to initialize itself and establish a communication link with the application used to access the external data. For example, the PDF-2 database DEO uses the following *initializeAccess* method to create and initialize a local cache and establish communication with the socket-based server for the PDF-2 database:

SmalltalkDB Fragment 4.6 PDF-2 DEO "initializeAccess" Method

```
initializeAccess
    "Create a new socket.  Create a cache if it doesn't exist (we
    want to keep the old ones if they do exist as they may have 'hot' data
    in them)"
    UserGlobals at: #_PDFDatabaseSocket put: (GsSocket new).
    (UserGlobals includesKey: #_PDFDatabaseCache) ifFalse: [
        UserGlobals at: #_PDFDatabaseCache put:
              (Cache new: 1000 key: #[-1,-1] data: ((String new) size: 4000))].
    "Try to connect to the server.  If we fail, we'll close the socket so reads and
    writes will fail as well"
    ((UserGlobals at: #_PDFDatabaseSocket) connectTo: 54321 on: 'lucy.cse.ogi.edu')
    ifTrue: [^true]
    ifFalse: [
        (UserGlobals at: #_PDFDatabaseSocket) close.
        ^false.
    ]
```

An *initializeAccess* method must be called for each DEO that encapsulates an external data source a user might access. The effect of forgetting to initialize a particular DEO is that all data access will fail and the DEO will answer "nil" to all attribute requests. Also, notice the use of "UserGlobals" as a location for the cache and socket in the example above. In GemStone, each user has a "UserGlobals" name space. By placing the cache and socket objects in a user's name space, DEO methods will access a user's private cache and socket objects. Concurrency conflicts between users over the cache and socket objects are thus eliminated since they are not shared among users.

While a DEO is theoretically comprised of three layers, an interesting modeling problem has arisen in our materials science schema that has forced us to augment the Database

Encapsulation Layer. The problem is that the class *Crystal* has two attributes—*unitCell* and *reducedUnitCell*—that are both instances of the *UnitCell* class. As such, both the *unitCell* and *reducedUnitCell* will forward identical messages on to the DEO when they need to retrieve data (e.g., *volume: aKey*). From the message signatures alone, the DEO is unable to determine whether to return the attributes of the unit cell or reduced unit cell.

One potential solution to this problem is to simply create a separate *ReducedUnitCell* class and use different message signatures (i.e., *UnitCell* with *a*, *alpha*, etc. and *ReducedUnitCell* with *reducedA*, *reducedAlpha*, etc.). However, this approach causes the Schema Layer to reflect a modeling problem at a lower level of the architecture and obscures the fact that a *reducedUnitCell* attribute holds the same sort of object as the *unitCell* attribute. It also requires us to maintain separate but equivalent class definitions and method code.

The general solution we have chosen to implement places pseudo-DEO objects between the Schema Layer objects and the DEO to resolve naming conflicts. The pseudo-DEO objects have different classes that correspond to the separate entities stored in the external data source. As depicted in Figure 4.2, the pseudo-DEO objects for the *unitCell* and *reducedUnitCell* attributes provide the same interface to Schema Layer *UnitCell* class objects. However, each sends a slightly different message along to the DEO (*aKey at: 'D'* vs. *aKey at: 'E'*).

4.3.3 External Data Source Layer

As we mentioned in Section 4.2.2, the OOHDB currently provides access to 5 external data sources—the two NIST databases as well as collections of Desktop Microscopist, CIF, and CAChe files.

In order to make these data sources accessible to the OOHDB, we have developed two platform-independent BSD socket-based "servers" that run outside of GemStone. Servers are run on each machine that hosts data sources of interest to the OOHDB. The multi-threaded servers are capable of interacting with multiple OOHDB sessions concurrently by forking processes to service each OOHDB user.

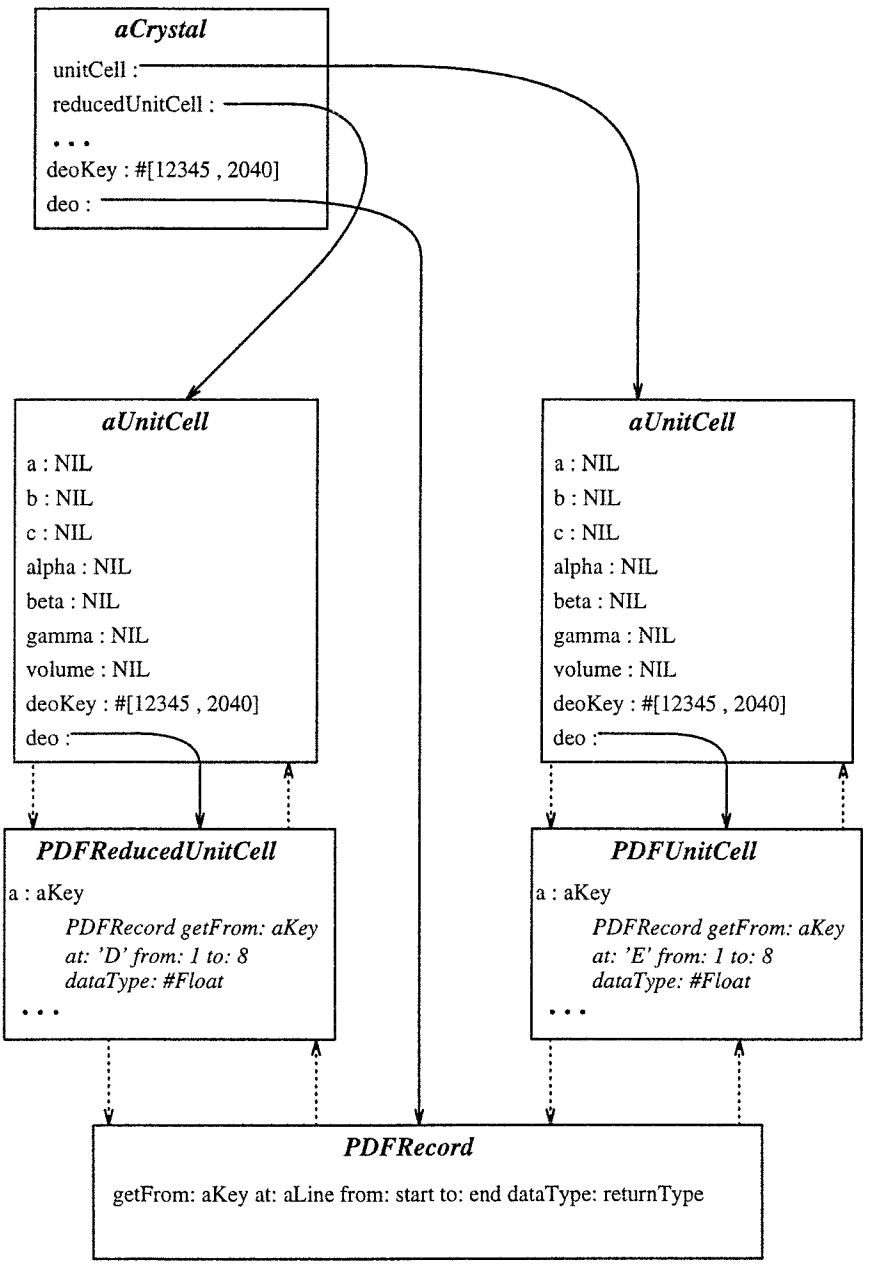


Figure 4.2: Pseudo-DEOs for Name Conflict Resolution

A simple "byte-server" is used to access the two NBS databases (see Program D.1 in Appendix D). Multiple byte-servers run concurrently on the Macintosh, each accessing a single database. The byte-servers respond to requests to read a given number of bytes from a database file starting at a specified offset (e.g., "r 10385620 2040").

The second server is a nearly identical "file-server" that is used to access the Desktop Microscopist, CIF, and CAChe files (see Program D.2 in Appendix D). A single file-server is run on each host that stores files accessible via the OOHDB. The file-server responds to requests to read an entire file (e.g., "r /ogi/students/dhansen/CIF/Al.cif").

These two servers have been designed to be general enough for use with many similar data sources. Although we do not currently access data managed by a DBMS, we anticipate that a similarly generic "DBMS-server" could be developed that executed simple queries for retrieving entities from the DBMS.

4.4 Objects, DEOs, and Databases—An Example Query

Now that we have described the architecture and provided examples of the processing involved in accessing external data, we turn to a concrete example to demonstrate how the various layers interact and the overall query performance of our architecture.

Our materials science OOHDB focuses on the *Crystal* class. Users of the OOHDB query the named GemStone collection *AllCrystals* that contains approximately 206,000 *Crystals*. All but a few *Crystals* are drawn from the two NBS CD-ROM databases. A handful of *Crystals* are stored in Desktop Microscopist and CAChe data files. One *Crystal* is available in a CIF file[13].

We use the following simple query for most of the following discussion and performance analysis:

SmalltalkDB Fragment 4.7 Example Query

```
AllCrystals select: [:c | c spaceGroup = 'R3m']
```

[13]The lack of CIF-based *Crystal* objects is due to the absence of CIF files in general. Though recently proposed as a data interchange standard, it appears that CIF has yet to be widely adopted.

This query iterates over the *AllCrystals* collection, sending each *Crystal* the *spaceGroup* message. The query returns a collection of the *Crystal* objects with a matching spaceGroup attribute value.

Figure 4.3 depicts the processing that takes place when a *Crystal* receives the *spaceGroup* message. There are a maximum of five (5) message sends to retrieve an attribute value from an external data source for a single *Crystal*:

1. The *spaceGroup* message to the *Crystal.*
2. The *Crystal* forwards the *spaceGroup:* message to the DEO with the *Crystal*'s *deoKey.*
3. The DEO sends itself the *getFrom:at:from:to:dataType:* method to extract and convert the *spaceGroup* value from the data.
4. The *getFrom:at:from:to:dataType:* method in turn sends the DEO the *getData:* message to retrieve the entire data record from the external data source.
5. The DEO *getData:* method sends a message to the byte-server to retrieve the data record.

Our layered implementation completely hides the details of accessing external data sources. However, this mechanism does not hide the latency of accessing external data sources from the user.

4.4.1 Assessing Performance

We have used Fragment 4.7 to test the performance of our architecture for accessing external data sources.

Figure 4.4 depicts the *target* hardware and software environment. During development and testing, Desktop Microscopist, CAChe, and CIF files were temporarily located on Smoked where a single file-server process provided access to those files.

The hardware configuration consists of two Sun SPARCstation II[14] workstations,

[14]SPARCstation is a registered trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems.

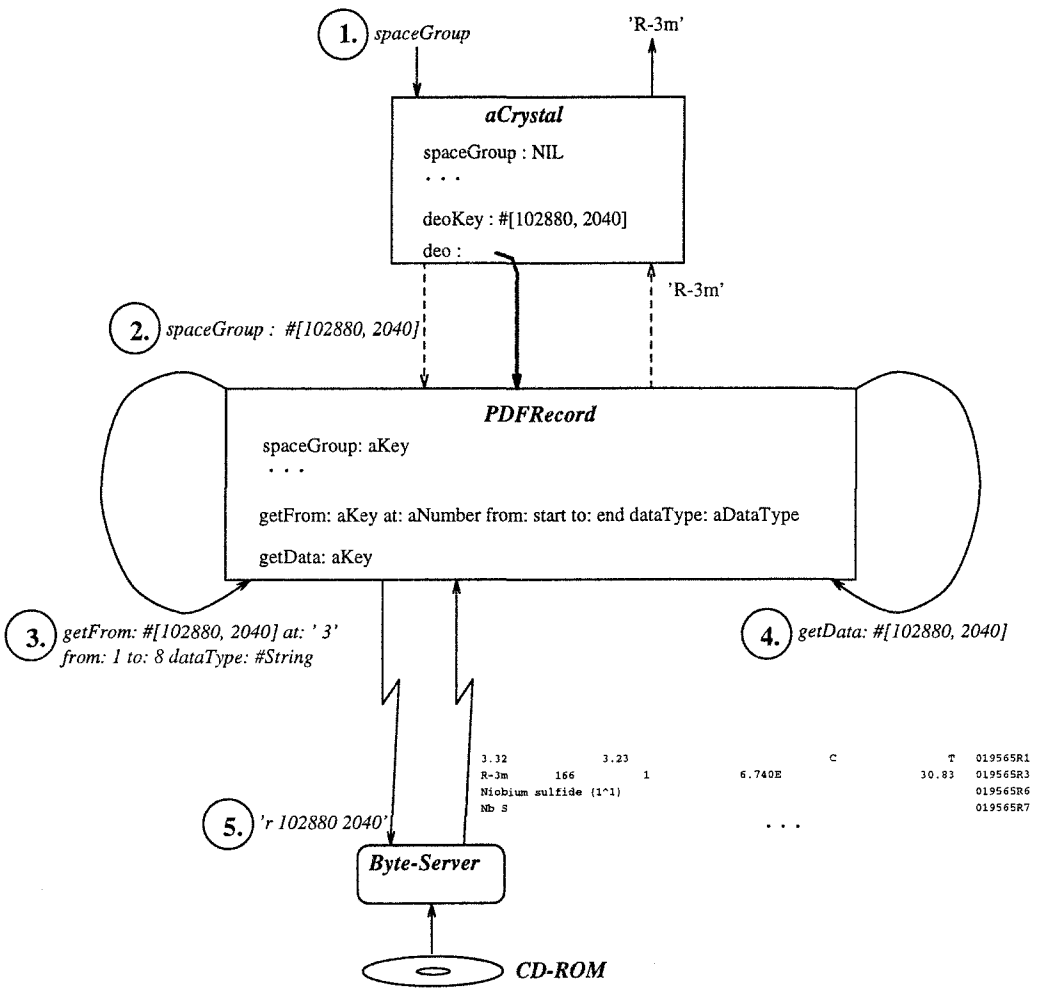


Figure 4.3: Responding to the *spaceGroup* Message

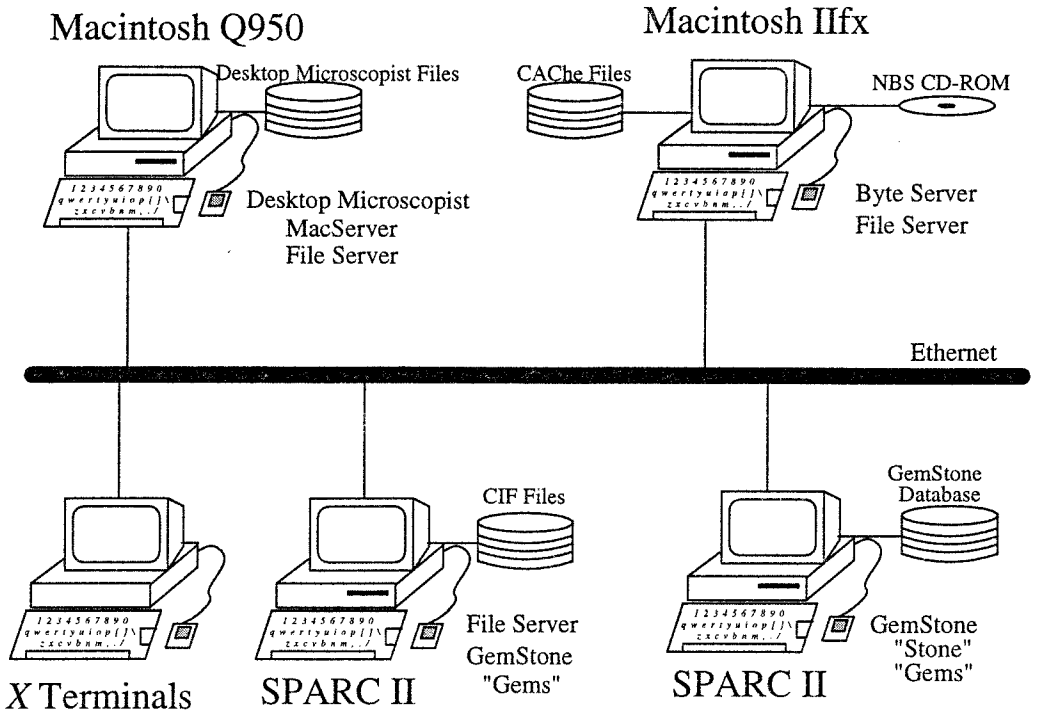


Figure 4.4: Hardware and Network Configuration

"Coho" and "Smoked", running SunOS4.1.3. Each workstation has 48MB of main memory. We are currently running v4.0.1 of the GemStone OODBMS. The GemStone server ("stone") runs on Coho and any of the various front-end clients ("gems") (i.e., topaz, GeODE, GemStone Smalltalk, C, C++ Interfaces) may be run on either Smoked or Coho and accessed via any X-Terminal. File-server processes (file-server C Program D.2) are run on Smoked and Coho to provide access to the Desktop Microscopist, CACheFiles, and CIF files stored on those systems. Two Macintosh[15] systems, "Lucy" (a IIfx) and the unnamed Q950, provide access to the two CD-ROM databases and support for the Desktop Microscopist application program. Lucy hosts the NBS CD-ROM using an Apple CD300e double-speed CD-ROM drive with an average seek time of 295 milliseconds and a peak transfer rate of 300k bytes per second. Two byte-server processes (byte-server

[15]Macintosh is a registered trademark of Apple Computer.

C Program D.1[16]) run in the background on Lucy to provide access to the two databases, Crystal and PDF-2, contained on the NBS CD-ROM. BSD sockets[17] are used for communication between GemStone and the byte and file-servers.

All tests were conducted during periods when the systems were otherwise lightly-loaded.

We begin by reporting the data access performance of the OOHDB where the attributes requested are not retained by the Schema Layer objects nor is any raw data cached by DEOs in the Database Encapsulation Layer. This test examines the worst-case performance of the architecture where all attribute values must be retrieved from the External Data Source Layer of the architecture. The performance of this test is poor enough that we ran Fragment 4.7 over a subset of 2,000 *Crystals* drawn from the *AllCrystals* collection of 206,000 *Crystals*. Table 4.1 shows the results of running Fragment 4.7 on the 2,000 *Crystal* subset.

Table 4.1: Un-optimized Query Performance (in seconds)

Run	*Elapsed Time*	*GemStone CPU*
1	1168	51
2	1152	48
3	1142	47
4	1155	48
Average	1154	48

The large disparity between elapsed clock time and GemStone CPU usage in Table 4.1 suggests that GemStone is idle for most of this query. The vast majority of the time is being spent accessing data from the CD-ROM databases via the network. In fact, performance is particularly poor since data is requested from the external data sources in a random order. Since the objects in *AllCrystals* are not sorted by their location in external data sources, we expect that we are likely paying the average 295 milliseconds of seek time overhead

[16]The byte-servers running on the Macintosh are slightly modified versions of C Program D.1 since the Macintosh does not support forking child processes.

[17]The implementation of BSD sockets for the Macintosh is provided by the Grand Unified Socket Interface (GUSI), a freeware package developed and supported by Matthias Neeracher.

each time we access the CD-ROM. Combined with the small size of the records on the CD-ROM (2k-4k bytes), each CD-ROM access thus takes approximately 300 milliseconds of set-up and read time—the seek time thus being a major source of overhead. This random access pattern negates the benefit of CD-ROM drive and filesystem buffers. One unexplored optimization might be to sort the *AllCrystals* collection by *deo* and *deoKey* to improve the performance of accessing external data sources, making better use of buffers by accessing data in the order it is stored on the disk. However, it is not clear that the effort required to sort the *AllCrystals* collection would substantially improve the performance. Since we expect most queries to minimize external access by querying retained attributes and cached raw data first, the few objects that are selected and require external access are unlikely to be co-located in the external databases.

What is clear is that this level of performance is unacceptable. Fortunately, our architecture and GemStone provide a number of mechanisms for minimizing external access and optimizing query performance.

4.5 Making it Go Fast

In order to optimize the performance of our materials science OOHDB we must be able to identify and optimize commonly used access paths. Since we can not hope to optimize every arbitrary query, our goal is to optimize those paths typically accessed by users to make broad first cuts at the data. Materials scientists typically use chemistry, d-spacings, space group, unit cell volume, etc. to reduce the search space to a small size. By optimizing these sorts of query paths we can provide performance that approaches the optimized performance of the underlying OODBMS for queries over the OOHDB as a whole.

The two principle optimization mechanisms we employ are caching data from external data sources in the Database Encapsulation Layer to minimize external access, and retaining attribute values in the Schema Layer and building indexes to speed up query processing.

4.5.1 Caching Data

Caching raw data is a simple and powerful mechanism for improving the performance of our architecture. If the DEO has cached the raw data record from a previous read, the DEO can extract and convert the attribute value from the cached data, eliminating external data access. The processing depicted in Figure 4.3 is reduced to three messages:

1. The *spaceGroup* message to the *Crystal.*

2. The *Crystal* forwards the *spaceGroup:* message to the DEO with the *Crystal*'s *deoKey.*

3. The DEO uses its *getFrom:at:from:to:dataType:* method to extract and convert the *spaceGroup* value from data in the cache.

The caches for the NBS CD-ROM database DEOs are managed using a FIFO replacement policy. The cache is composed of two parts: a *keys* attribute based on the *SortedCollection* GemStone class providing $O(\log_2 n)$ access using a binary probe, and a *data* attribute based on the GemStone *Array* class that holds the cache lines. We separate the keys from the data for two reasons. First, it improves the performance of maintaining the cache since only the keys must be reordered when a cache line is replaced and second, it allows us to manage the cache using a FIFO cache-replacement policy that would be difficult to implement if the cache were reordered each time a line were replaced.

We tested the effectiveness of DEO caches by executing Fragment 4.7 twice in succession. The first time the query was executed, the DEO caches were empty and the results were as presented earlier in Table 4.1. A side effect of this first execution was that the caches were filled with the raw data for the 2,000 *Crystal* subset. Thus the second execution of the query accessed the raw data from the cache, eliminating all external data access. The caches were then emptied and this pair of tests re-run several times. Table 4.2 and Figure 4.5 compare the results from Table 4.1 with empty caches to the same query over the same 2,000 *Crystal* subset with the caches full.

As we would expect, the results in Table 4.2 and Figure 4.5 show that GemStone's CPU usage is a much greater percentage of the overall clock time now that the network and CD-ROM latency have been removed.

Table 4.2: Performance With Caching (in seconds)

	Cache Empty		*Cache Full*	
Run	*Elapsed Time*	*GemStone CPU*	*Elapsed Time*	*GemStone CPU*
1	1168	51	22	11
2	1152	48	21	13
3	1142	47	27	11
4	1155	48	25	13
Average	1154	48	24	12

The performance improvement achieved by eliminating external data access is dramatic and suggests that maintaining large fixed-size caches within the DEOs is a very effective tuning mechanism. The use of DEO caches will be most effective in cases where users execute queries over the *AllCrystals* collection to select a subset that they will work with for some time. If such a "working set" fits within the DEO caches, we eliminate all external access for subsequent queries over the working set.

4.5.2 Retaining Attribute Values

Retaining attribute values, such as the *spaceGroup*, in each *Crystal* provides yet another dramatic performance increase. An OOHDB query that accesses retained attributes behaves like an ordinary OODBMS query—the Schema Layer objects simply return the value of the attribute in response to the query. There is no interaction with the Database Encapsulation or External Data Source Layers of the architecture at all. The processing depicted in Figure 4.3 is reduced to one message:

1. The *spaceGroup* message to the *Crystal.*

Accessing retained attribute values provides a level of performance that allows us to query the entire collection of 206,000 *Crystals* in reasonable time. Table 4.3 presents the results of running Fragment 4.7 over the entire collection of 206,000 *Crystals* with the *spaceGroup* attribute having been previously accessed and retained by each of the *Crystals.*

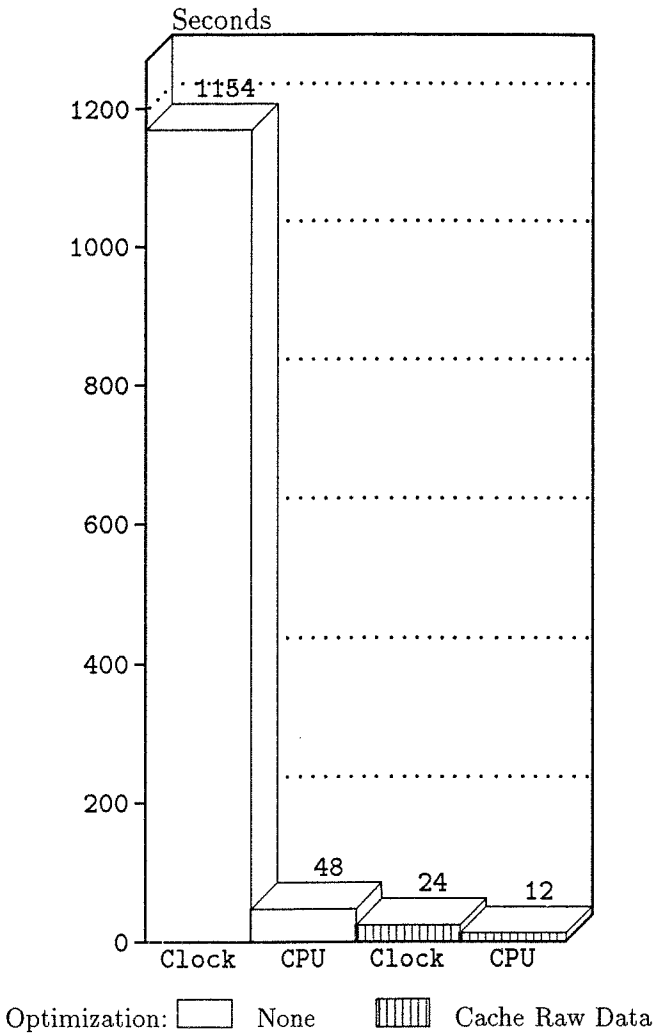


Figure 4.5: Performance Chart for Caching

The performance boost here is derived from eliminating the message overhead of communicating with the DEOs, the $O(\log_2 n)$ search of the DEO caches, and the extraction and conversion of raw data. Querying retained attributes is equivalent to querying the underlying OODBMS in its native mode. While retaining attribute values is an effective tuning mechanism, an even greater benefit of retaining attributes is that indexes can be constructed for them to provide optimized query performance.

Table 4.3: Performance with Retained Attributes (in seconds)

Run	*Elapsed Time*	*GemStone CPU*
1	159	51
2	154	53
3	158	54
4	155	53
Average	157	53

4.5.3 Indexing

We retained the *spaceGroup* attribute for each of the 206,000 *Crystals* in the OOHDB and then built an index for the collection *AllCrystals* using this attribute. Table 4.4 and Figure 4.6 compare access using an index to the results from Table 4.3 where retained attributes were accessed.

Table 4.4: GemStone Optimized Query Performance

	Attribute Access		*Using Index*	
Run	*Elapsed Time*	*GemStone CPU*	*Elapsed Time*	*GemStone CPU*
1	159	51	8	7
2	154	53	8	7
3	158	54	12	6
4	155	53	12	6
Average	157	53	10	7

The use of indexes provides yet another leap in performance. Queries using OODBMS indexes are native, optimized OODBMS queries. Furthermore, indexing should scale well ($O(\log_2 n)$) as the size of the OOHDB grows. The only drawback to the use of indexes is that Fragment 4.7 must be modified slightly in GemStone:

SmalltalkDB Fragment 4.8 Example Query (optimized form)

```
AllCrystals select: {:c | c.spaceGroup = 'R3m'}
```

The use of the query delimiters "{}" and the "dot notation" (i.e., c.spaceGroup) are used in GemStone to indicate that the query processor should bypass the method interface and

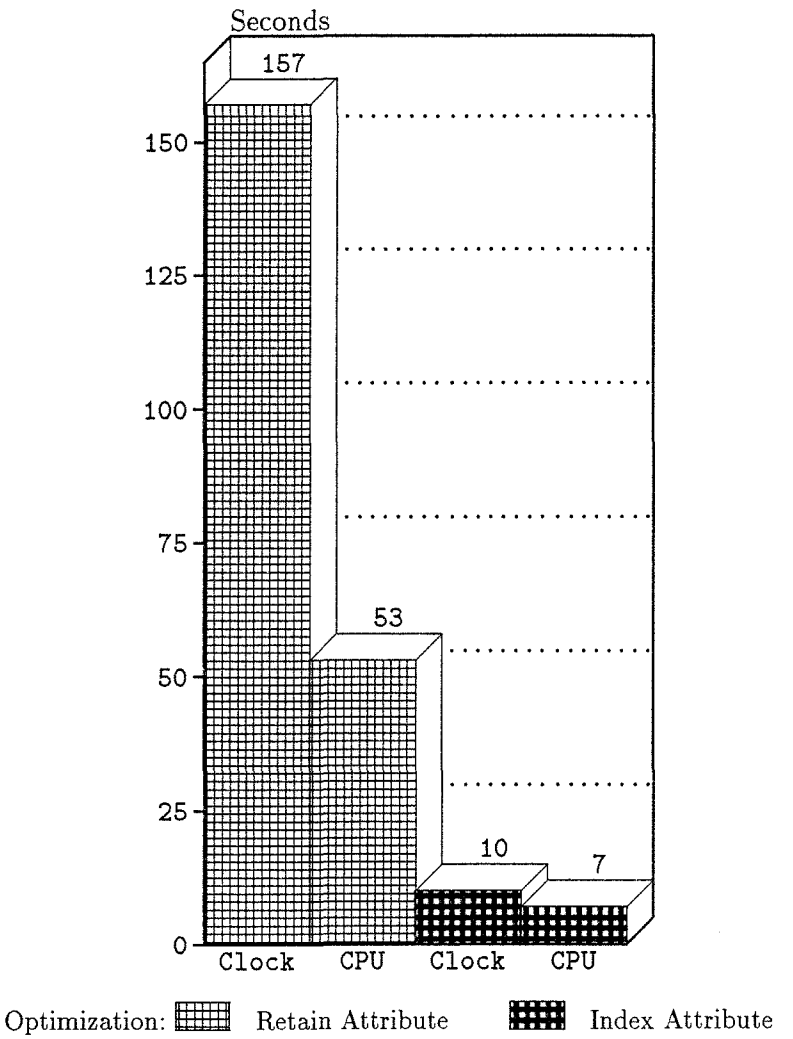


Figure 4.6: Performance Chart for Indexed Access

use structural access instead, together with indexes if they exist.

However, if the *spaceGroup* attribute has *not* been retained by all *Crystal* objects, then the answer to Fragment 4.8 will be completely different than the answer to Fragment 4.7. Accessing objects via the method interface, Fragment 4.7 will return the proper answer, albeit slowly, while Fragment 4.8, bypassing the method interface and accessing the attribute values directly, will return an empty set since for each *Crystal* the attribute value "nil" will fail to match 'R3m'. While this divergent behavior may appear to be a serious drawback, the fact is that most OODBMSs require queries to "break encapsulation" and resort to structural access in order to use optimizations such as indexes. Any time

the method interface to an attribute does more than simply return the attribute value, queries that use methods can not be considered semantically equivalent to queries that use structural access.

4.5.4 Performance Summary

We have presented performance results that explore four levels of performance: un-optimized, cached raw data, retained attributes, and indexed access. The first two levels were examined using a subset of the *AllCrystals* collection, while the last two levels were examined by querying the entire collection.

In order to draw a comparison between all four levels of performance, we use the average times reported in Table 4.2 to estimate the theoretical performance of our un-optimized and raw data cache queries for the entire set of 206,000 *Crystals* using a theoretical cache capable of holding all 206,000 *Crystals*. For the most part, our estimates can assume linear behavior. However, the DEO caches are searched using a simple binary probe so the time required to search a cache of size m for n *Crystals* will grow as $n \log_2 m$. The time spent searching the cache *is* significant and so we must account for this non-linear behavior in order to derive accurate estimates. We have "profiled" the performance of Fragment 4.7 with the DEO caches empty and full to identify the portion of GemStone CPU time spent searching the cache. If we assume that the DEO caches are capable of holding all of the data (i.e., $n = m$), then we can estimate the amount of CPU time GemStone will spend searching DEO caches with the following equation:

$$(sizeof(AllCrystals)/sizeof(CrystalSubSet))*$$
$$((CPUTime - CPUCacheSearchTime)+$$
$$CPUCacheSearchTime * \log_2(sizeof(AllCrystals)/sizeof(CrystalSubSet)))$$

With the cache empty, Fragment 4.7 spends 10.79% of the 48 seconds of GemStone CPU usage searching the cache for a total of 5.2 seconds. Thus, our estimate of the GemStone CPU usage required for querying a collection of 206,000 *Crystals* with no cache hits on a theoretical 206K-line cache is:

$$(206/2) * ((48 - 5.2) + 5.2 \log_2(206/2)) = 7990 \; seconds$$

In estimating the total elapsed clock time for this query, we note that the vast majority of the time is spent communicating with the external database. That portion of the elapsed clock time should scale linearly. So to estimate the elapsed clock time we compute the time for the linear, non-CPU portion and add the GemStone CPU usage computed above:

$$((1154 - 48) * (206/2)) + 7990 = 121908 \; seconds$$

With the cache full, Fragment 4.7 spends 18.90% of the 12 seconds of GemStone CPU usage searching the cache for a total of 2.3 seconds. Thus, our estimate of the GemStone CPU usage required for querying a collection of 206,000 *Crystals* with 100% cache hits on a theoretical 206K-line cache is:

$$(206/2) * ((12 - 2.3) + 2.3 \log_2(206/2)) = 2583 \; seconds$$

Again, we estimate the elapsed clock time by assuming that non-CPU time scales linearly:

$$((24 - 12) * (206/2)) + 2583 = 3819 \; seconds$$

Figure 4.7 presents a log plot summary comparing the estimated performance for un-optimized and DEO cache queries computed above to the actual results from Table 4.4 for attribute retention and indexed query access.

One notable feature of Figure 4.7 is that each of the optimizations we have discussed provides an improvement of at least one decimal order of magnitude over lower states of optimization. The degree of improvement at each level of optimization suggests that each optimization has a place in the OOHDB architecture. Since we cannot cache all raw data or retain all attributes and construct indexes over them, we expect that a particular OOHDB will make judicious and appropriate use of each optimization mechanism to provide the optimal level of performance in a reasonable amount of space. For our materials science OOHDB this means retaining a few commonly queried attributes, such as *spaceGroup* and *maxdSpacing*, as well as using large DEO caches.

The other notable feature of the results in Figure 4.7 is the large affect that accessing external data sources has on performance. External access must clearly be minimized in order to achieve reasonable performance. Fortunately, the tuning mechanisms we have

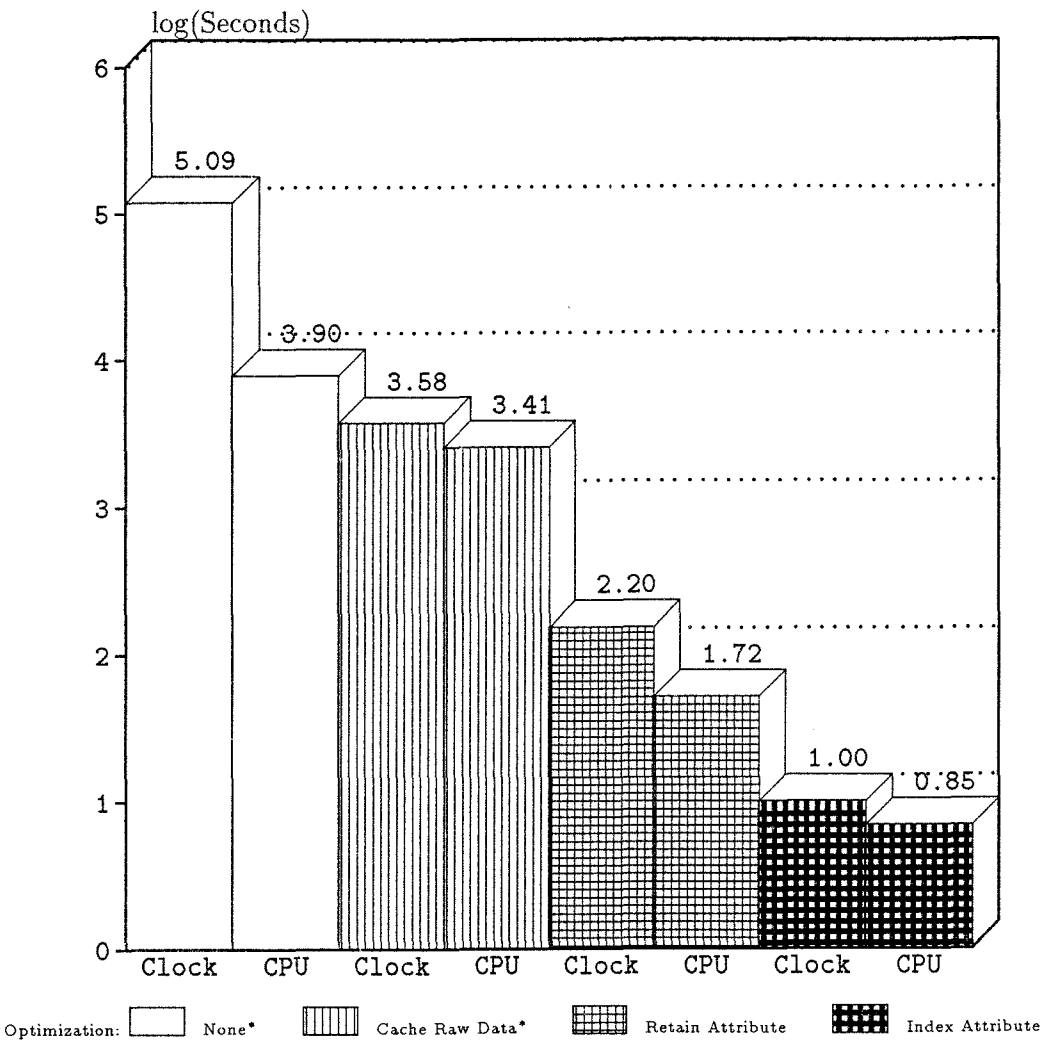


Figure 4.7: Estimated* and Actual Performance for 206,000 Crystals

presented here provide reasonable and effective mechanisms for minimizing costly external access.

4.5.5 Alternative Access Structures

In addition to built-in indexes, we have devised additional "alternative access structures" for optimized access to the data in the OOHDB. The purpose of these structures is to provide alternative optimization mechanisms in cases where indexes cannot be used or are not efficient.

One alternative access structure is the *CrystalByElement* "dictionary". The *CrystalByElement* dictionary is designed to give optimized access to subsets of *AllCrystals* based on the chemistry of the *Crystal*. For each element symbol (e.g., Al), *CrystalByElement* stores a subset of *AllCrystals* containing all *Crystals* where the chemical formula includes that element. These subsets comprise a set of pre-computed queries for each *Element* of the form:

SmalltalkDB Fragment 4.9 Element Query

```
AllCrystals select: [:c | (c molecule formula atoms)
                    detect: [:a | a element symbol = #Al]]
```

Initially, *CrystalByElement* was developed because an earlier version of GemStone did not permit index paths to contain multi-valued objects (e.g., a *Set*) in the path. Thus we were unable to build an index for *AllCrystals* using the path molecule.formula.atoms.-element.symbol because "atoms" is a set of *Atoms*. Though version 4.0 of GemStone *does* permit multi-valued objects in an index path, GemStone still builds an index for each of the intermediate steps in the index path. So, for the path molecule.formula.atoms.element.-symbol, five indexes are constructed. Since we are only interested in accessing *Crystals* by element symbols, the intermediate indexes (i.e., molecule, molecule.formula, molecule.-formula.atoms, molecule.formula.atoms.element) are irrelevant and somewhat wasteful.

The following query is an example of how we might query *AllCrystals* looking for *Crystals* with Aluminum and Copper in their formula:

SmalltalkDB Fragment 4.10 Query *AllCrystals* by Element

```
AllCrystals select: [:c | ((c molecule formula atoms)
     detect: [:a | a element symbol = #Cu]) and: [
        (c molecule formula atoms) detect: [:a | a element symbol = #Al]]]
```

In contrast to Fragment 4.10, selecting a subset of *AllCrystals* for a given element using the *CrystalByElement* structure can be achieved in constant time based on the number of elements. This performance is the result of pre-computing the subsets and sequentially searching the *CrystalByElement* dictionary for the appropriate subset. The real power of this structure is that disjunctive and conjunctive queries, such as that in Fragment 4.10, can be written using set intersect and union operations that have linear $O(max(n, m))$

performance where n and m are the cardinality of the sets. This linearity occurs because GemStone maintains collections sorted by OID allowing GemStone to union or intersect collections by merging the sorted collections. So we can rewrite Fragment 4.10 using the *CrystalByElement* structure and set intersection operations:

SmalltalkDB Fragment 4.11 Query using *CrystalByElement*

```
(CrystalByElement at: #Cu) * (CrystalByElement at: #Al)
```

Fragments 4.10 and 4.11 select 387 of the 206,000 *Crystals* in the OOHDB. Table 4.5 shows the performance improvement between Fragment 4.10 and the set-based Fragment 4.11. These sort of pre-computed queries are useful primarily where a large collection can be

Table 4.5: Performance Using the *CrystalByElement* Structure (in seconds)

	Querying AllCrystals		*Using CrystalByElement*	
Run	*Elapsed Time*	*GemStone CPU*	*Elapsed Time*	*GemStone CPU*
1	2878	1285	3	1
2	2717	1224	2	1
3	2711	1230	1	1
4	2721	1231	2	1
Average	2757	1243	2	1

partitioned into possibly overlapping subsets using a discrete variable. *CrystalByElement* stores 98 overlapping subsets of the *AllCrystals* collection—one subset for each of 98 chemical elements that are present in the crystals.

A second alternative access structure is *CrystalByVolume*, based on the GemStone *SortedCollection* class. The *CrystalByVolume* collection holds (*unitCell.volume*,*crystal*) pairs—one for each of the 206,000 *Crystals*. This structure essentially forms an index for *AllCrystals* using the *volume* attribute of the *unitCell*. *CrystalByVolume* is searched using a simple $O(\log_2 n)$ binary probe. The obvious question is why not retain and build an index over the *volume* attribute? The reason we use this alternative access structure is to minimize the space used in the OOHDB. As we noted in Section 4.3.1, we delay the creation of many of a *Crystal's* sub-objects, such as the *unitCell*, to minimize the space used in the OOHDB. If we were to build an index over the *volume* attribute of

the *unitCell* we would have to instantiate the *unitCell* sub-object for every *Crystal*, which would allocate some space for all of the other attributes of the *unitCell*. While instantiating *unitCell* objects and retaining the *unitCell.volume* attribute value may or may not pose a space problem, we used this opportunity to examine the effectiveness of such an alternative access structure.

Fragment 4.12 uses the *CrystalByVolume* structure to retrieve the set of *Crystals* that have a *unitCell volume* within a specific range (similar methods for equality, less-than, and greater-than are also defined):

SmalltalkDB Fragment 4.12 Query using *CrystalByVolume*

```
CrystalByVolume retrieveRange: (CrystalByVolume findRangeInclusive: #[999.0,1004.0])
```

Table 4.6 demonstrates that Fragment 4.12 performs similarly to the index-based query reported in Table 4.4 where we observed an average of 10 seconds of elapsed time and 6 seconds of GemStone CPU usage. This similarity is to be expected since the *CrystalByVolume* structure behaves much like an index.

Table 4.6: Performance Using the *CrystalByVolume* Structure (in seconds)

Run	*Elapsed Time*	*GemStone CPU*
1	13	3
2	11	2
3	11	3
4	10	2
Average	11	3

These two alternative access mechanisms demonstrate the potential for constructing special-purpose query evaluation mechanisms. Of course it remains up to the users of the OOHDB to use these structures in their queries. New structures can similarly be built to optimize other important common access paths.

4.6 Populating and Maintaining the OOHDB

In Section 3.4 we noted that each DEO should also include methods to populate and maintain the OOHDB. There are two aspects to population and maintenance. The first is determining what objects are present in the external data source and the second is populating the OOHDB with empty Schema Layer objects.

In the case of the PDF-2 database DEO, we determine where records begin and end in the database by scanning the file looking for a '1' in the 80th character of each 80-column card—indicating that this is the first card of a new record. The following *getKeysStartingAt:* method uses the same *getData* method presented earlier to scan the database 80 bytes at a time, detecting where records begin, and building an array of keys for the records in the database:

SmalltalkDB Fragment 4.13 PDF-2 DEO "getKeys" Method

```
getKeysStartingAt: start
    "get the offsets into the file"
    | keys string i done |
    self initializeAccess.
    keys := Array new.
    i := start.
    done := false.
    "Start reading 80-byte chunks from the server.  Each time byte 80
    holds a '1' we have the beginning of a new record.  For each record
    add a 2-element array to 'keys' with the offset and the number of
    bytes for this record."
    [done] whileFalse: [
        "Get 80 bytes from the server - we're done if less than 80 get
        returned."
        string := self getData: #[i, 80].
        (done := (string size < 80) ) ifFalse: [
            string at: 80 = $1 ifTrue: [
                "Patch the size of the last record"
                keys size > 0 ifTrue: [
                    (keys last) at: 2 put: (i - ((keys last) at: 1))].
                "Add this new offset to the array"
                keys add: #[i, 0] ].
            i := i + 80 ] ].
    "Patch the last key's number of bytes"
    keys size > 0 ifTrue: [(keys last) at: 2 put: i].
```

```
^keys
```

This method accepts a single parameter that determines the start position of the scan. This parameter is useful for maintaining the OOHDB since the PDF-2 database is updated by appending data. Thus new releases of the PDF-2 database can be scanned quickly by beginning with the last known record and scanning forward.

The *buildCrystals:* method that populates the OOHDB with *Crystal* objects for each record identified in the PDF-2 database uses the *getKeysStartingAt:* method. The boolean parameter "update" is used to determine whether *buildCrystals:* scans the entire file, or scans for new data appended to the file:

SmalltalkDB Fragment 4.14 PDF-2 DEO Population Method

```
buildCrystals: update
    "Add new Crystal objects for each record in the external database"
    | keys key newCrystal max newMolecule empFormula chemFormula e c n k m |
    self initializeAccess.

    update ifTrue: [
        "Find the largest key for this database so far"
        max := AllCrystals detect: [:c | c deo == self].
        AllCrystals do: [:c | (c deo == self) and: [
             ((c deoKey) at: 1) > ((max deoKey) at: 1)] ifTrue: [
                    max := c] ].
        "Get the keys for the new data at the end of the file"
        keys := self getKeysStartingAt: ((max deoKey) at: 1) + ((max deoKey) at: 2)]
    ifFalse: [
        "Get all the keys in the file"
        keys := self getKeysStartingAt: 0].

    keys do: [:key |
        "Create a new Crystal with self as the DEO and using this key"
        newCrystal := (Crystal withDEO: self withKey: key).
        "Look for an existing Molecule with the same name and formula, if
        found, this Crystal will point to that Molecule"
        c := self chemicalFormula: key.
        n := self compoundName: key.
        newMolecule := AllMolecules detect: {:m | (m.name = n) &&
                    (m.formula.formula = c)} ifNone: [
            "Existing Molecule not found.  Create a new one"
            m := Molecule new.
```

```
        "Share formula objects if the chemical and empirical formulas
        are identical"
        chemFormula := Formula fromString: c.
        empFormula := Formula fromString: (e := self empiricalFormula: key).
        (c = e) ifTrue: [empFormula := chemFormula].
        m name: n.
        m formula: chemFormula.
        m empiricalFormula: empFormula.
        "Add this Molecule to the set of all Molecules"
        AllMolecules add: m.
        m].
    newCrystal molecule: newMolecule.
    "Add this new crystal to the set of all crystals and update the
    optimized access paths for that collection"
    AllCrystals add: newCrystal.
    newCrystal addToAccessPaths]
```

Together, *buildCrystals* and *getKeysStartingAt* provide the mechanism for populating and maintaining the OOHDB for the PDF-2 DEO. The initial population of our materials science OOHDB from the two NBS CD-ROM databases using the *buildCrystals* method takes approximately 48 hours of elapsed time—a predictable result given that in Section 4.5.4 we estimated it would take 121,908 seconds (approximately 34 hours) of elapsed time to query all 206,000 *Crystals* from the CD-ROMs with no cached or retained data. The added expense in populating the OOHDB is due to the fact that we are required to access the CD-ROM databases 80-bytes at a time in order to locate the beginning of each record. Some computational overhead is also incurred as we connect new *Crystal* objects to pre-existing *Molecules*. Fortunately, as we noted earlier, these CD-ROM databases are updated by appending new data. The design of our *buildCrystals* method thus provides much better performance during subsequent updates of the OOHDB.

Population and maintenance methods for the other DEOs are similar. The primary difference is that new objects are detected by scanning a directory looking for new files instead of scanning a single large file looking for the beginning of records.

4.7 Summary

In assessing our materials science OOHDB, we wish to revisit our primary goals—a simple-to-implement OOHDB that integrates lightly-managed data.

We believe that the implementation presented here demonstrates how simply the architecture can be constructed. The implentation required very little code development. Attribute accessing methods were each extended with a few additional lines of SmalltalkDB code. The implementation of DEOs was relatively simple as well. For example, the methods implementing the DEO for the PDF-2 database consist of approximately 300 lines of commented SmalltalkDB code. The two generic servers for bytes and files total 335 lines of commented C code. Alltogether, the amount of code writen to implement our GemStone-based materials science OOHDB totalled approximately 800 lines of commented GemStone SmalltalkDB code and 335 lines of commented C code.

The five data sources accessed by the materials science OOHDB are all in the form of lightly-managed data. The sources range from the highly-structured NBS CD-ROM databases to the context-sensitive single-file CIF format. As our performance numbers demonstrate, commonly used query paths can be optimized to achieve performance equivalent to the optimized performance of the underlying OODBMS. Alternative access structures are used to enhance performance where OODBMS optimizations are insufficient or in order to provide a more space-efficient optimization.

The materials science OOHDB we have presented here is a relatively straightforward GemStone implementation of our OOHDB architecture. In Chapter 5 we examine two popular commercial OODBMSs in light of our architecture and assess their suitability to serve as implementation tools for our OOHDB architecture.

Chapter 5

Architectural Portability

We assess the generality of our OOHDB architecture by examining two other popular[1] commercial OODBMSs for their suitability as tools for implementing our OOHDB architecture. This chapter briefly looks at the O_2 and ObjectStore OODBMSs. We examine these two databases in particular for two pragmatic reasons. First, we have access to—and at least limited experience with—both products. Second, characteristics of these two products are key elements in the definition of the *Object Database Standard* (ODMG-93) [Cat93] that attempts to specify a vendor-independent standard for object databases. Specifically, the data model and C++ language bindings of ObjectStore and the object-oriented O_2SQL data manipulation language of O_2 have, in large measure, been adopted for the ODMG standard. Thus, by exploring these two OODBMSs we implicitly address issues that are likely to have even wider applicability.

5.1 Required OODBMS Features

Our architecture imposes very few requirements on the underlying OODBMS. The requirements include:

An object-oriented data model encapsulating structure and behavior. The behavioral component is essential since it provides the mechanism for transparently accessing external data and mapping the data to the homogeneous OOHDB schema.

[1] GemStone, ObjectStore, and O_2 were the commercial OODBMSs covered by Communications of the ACM in a Special Section on Next Generation Database Systems [Cat91].

Where that behavior is stored (within the OODBMS or in OS files) or where it is executed (client or server) does not appear to be important.

A computationally complete data manipulation language. The power of the language is important if we are to be able to provide mappings from arbitrary external data representations to the homogeneous OOHDB schema.

Support for accessing external data sources (e.g., BSD sockets). Access to data stored in an external DBMS may also be provided by the OODBMS via a direct OODBMS-DBMS "gateway".

The ability to store and detect "null" attribute values. The transparency of our approach relies on our ability to forward messages to a DEO when the attribute value in the OODBMS is null.

A query language allowing method invocation. Invoking methods in queries is essential since we use attribute accessing methods to transparently access external data on demand. Structural access will be used where appropriate for optimized access.

These minimal requirements are met by both O_2 and ObjectStore and are likely to be met by many OODBMSs. Though our architecture uses inheritance and procedural programming languages, it is likely that other paradigms would be sufficient as well (e.g., conformance-based typing and functional programming languages). However, the architecture presented in Chapter 3 would need to be modified accordingly.

5.2 O_2

O_2, from O_2 Technology[2], is a commercial object-oriented database management system that evolved from object-oriented database research conducted in the late-80's by researchers at GIP Altaïr [LRV90, Deu91].

[2] O_2 Technology, 7 rue de Parc de Clagny, 78035 Versailles Codex, France

O_2 uses a client-server architecture with a single server, the O_2Engine, and multiple clients.

A database is logically broken into "schemas" and "bases". Each base is an implementation of a schema, but multiple bases may share a common schema. The owner of the base controls access to the data in that base. Persistence is by reachability with manual garbage-collection of non-reachable data. O_2 supports the creation of indexes over non-sequenceable collections (e.g. sets and bags) that have been stored in a base using an explicit name (e.g., *AllCrystals*).

The data model of O_2 is loosely based on C++ [ES90] with the data model divided between primitive types (e.g., integer, float, char) and structured types (e.g., set, list, tuple, class). The O_2Kit provides a small class library of definitions including *date* and *text* classes. The data definition language is O_2C, a C++-like language for specifying the structure of objects and the program code for methods. Applications can be developed using the O_2Tools graphical programming environment and the O_2Look GUI toolbox using O_2C. O_2C methods are compiled and stored within the database, then dynamically linked and executed on demand. In addition to O_2C, O_2 provides a non-procedural data manipulation language called O_2SQL that is limited to read-only queries (i.e., no "insert", "update", or "delete" syntax). O_2 also provides application programmer interfaces for C and C++.

For the most part, O_2 meets or exceeds the minimal capabilities required by our architecture. O_2 is similar to GemStone functionally and architecturally. The most notable difference is the shift from a Smalltalk-oriented data model and database language to a C++-oriented data model with data manipulation languages based on C and SQL.

Communicating with external data sources from O_2 via BSD sockets is accomplished by writing O_2C methods that create, read, and write to BSD sockets. Since O_2C is a superset of ANSI C, methods can utilize the full functionality of the C language including function libraries.

One difficulty with O_2 is that the type system is divided into structured types and primitive types. Primitive types, such as integer, float, and char, do not have object semantics and most importantly, there is no notion of a "null" value for a primitive type.

This poses a problem for our message forwarding approach since it depends on detecting that an attribute has a null value.

One potential solution to this null-value problem is to select and set aside some legal value for each primitive type to serve as a null value. For example, we may decide to use the largest representable negative integer value as "null" for integer attributes. The obvious drawback of this approach is that both the methods in the OOHDB and the users of the OOHDB must be aware of the value chosen lest a null value be misinterpreted as real data.

A second approach is to wrap all primitive types in a structured type that does have a null representation. For example, we might define a class *Integer* where an instance holds nothing but an integer value. Objects of the *Integer* class would either have to be able to respond to all operations normally performed over integer values, or else all operations involving *Integer* objects would have to be coded in such a way that they operated over the integer value *inside* the *Integer* object (e.g., $*i + *j$). However, when operating on the values stored inside *Integer* objects, care must be taken to re-wrap the integer result within a new *Integer* object (e.g., $*k = *i + *j$). So, while this approach avoids the potential for confusing a null value with real data, it requires some discipline on the part of the database method developers and users to constantly wrap primitive types within objects and to carefully maintain that wrapping. This approach was used in the construction of an O_2 prototype of our materials science OOHDB that was a port of our early GemStone OOHDB [HM94]. The port was functional in less than two weeks of single-programmer effort.

Wrapping primitive types in objects proved to be a workable solution to the lack of null values for primitive types. So, for example, we declared a class *Integer* in O_2 that is of "type" integer:

OODBMS Fragment 5.1 O_2 Primitive Type Wrapper Class Definition

```
class Integer inherit Object public type
   integer
end;
```

The O_2 type system allows this sort of class declaration tying a class to a single type.

The integer value of an object of class *Integer* are accessed using C-like pointer syntax (i.e., *i). Using this approach we were able to create Schema Layer methods that forward messages on to DEOs after testing the value of the attribute:

OODBMS Fragment 5.2 O_2 Attribute Accessing Method

```
method body SpaceGroup: String in class Crystal
{
   if (self->spaceGroup == nil) && (self->deo != nil)
      if (virtualObjectCaching)
         return self->spaceGroup = self->deo->spaceGroup(self->deoKey);
      else
         return self->deo->spaceGroup(self->deoKey);
   else
      return self->spaceGroup;
}
```

Method 5.2 is a straightforward port of SmalltalkDB Method 4.2. Porting other elements of the architecture from GemStone to O_2 was similarly straightforward.

Query 5.3 is an example of an un-optimizable O_2SQL query that uses method-based attribute access similar to Query 4.7. Like Query 4.7, this query cannot be optimized by O_2 because it accesses the attribute using a method rather than direct structural access.

OODBMS Fragment 5.3 O_2 Example Query

```
select c from c in AllCrystals
   where c->SpaceGroup == "R3m"
```

Query 5.4 is an example of an optimizable query that uses structural access similar to GemStone Query 4.8:

OODBMS Fragment 5.4 O_2 Example Query (optimized form)

```
select c from c in AllCrystals
   where c.spaceGroup == "R3m"
```

Queries 5.3 and 5.4 have only minor syntactic differences. The use of "SpaceGroup" in Query 5.3 and "spaceGroup" in Query 5.4 is due to the fact that O_2 uses a common name-space for both methods and attributes within a class. As a result, whereas we had both an attribute and method called "spaceGroup" for our *Crystal* class in GemStone,

we must differentiate between the two in O_2—"spaceGroup" for the attribute name and "SpaceGroup" for the name of the method accessing that attribute.

Another aspect of O_2 that required a small modification to our GemStone implementation is the lack of private user data spaces within an O_2 database. As we noted in Section 4.3.2, we created persistent socket and DEO cache objects in each user's data space to avoid concurrency conflicts between users. In O_2, the problem of concurrent access to sockets and caches must be solved using a different approach since a "base" in O_2 is a monolithic data space shared by all users. Fortunately, O_2 provides "global variables" that solve the problem nicely. Concurrent access to sockets and caches is achieved by declaring global socket and cache variables at run-time. These global variables are not persistent and are private to a user, thus global variables are free of concurrency conflicts. Methods that access global variables declare them as "extern" and references to them are resolved at run-time. One drawback to this solution, however, is that global variables do not persist beyond the database session in which they were created. As a result, the contents of the caches are lost between sessions. One potential solution is for each user to write their caches to persistent uniquely named objects in the database before leaving O_2, then retrieve the caches into global variables when they initialize their next session.

The O_2-based OOHDB prototype we developed demonstrated that O_2 is suitable for implementing our OOHDB architecture. An O_2 OOHDB requires a few changes to the approaches used in GemStone, most notably in handling null values. In addition, where we were able to leverage GemStone's rich class hierarchy for building elements of the OOHDB infrastructure (e.g., *SortedCollection* for implementing caches, *Dictionary* for alternative access structures), O_2 requires the OOHDB implementer to develop these sort of support classes themselves.

5.3 ObjectStore

ObjectStore, from Object Design, Inc.[3], is a commercial object-oriented database management system based on the C++ programming language [Obj92, OHMS92, LLOW91].

[3]Object Design, Inc., Twenty Five Mall Road, Burlington, MA 01803

ObjectStore uses a client-server architecture where client applications request database pages from the server in response to page faults generated by the application.

ObjectStore uses a "Directory Manager" that provides a hierarchical organization of databases and objects within databases. This "directory" structure is used both to provide a name-space for persistent data and to control access to data via permission mode information. Persistence in ObjectStore is requested explicitly at object creation time using an overloaded *new()* function that takes a C++ pointer to an ObjectStore database as a parameter indicating the database in which the object is to be stored. Objects are deleted from a database using an overloaded *delete()* function. This form of explicit persistence is in contrast to the implicit persistence-by-reachability used by GemStone and O_2. ObjectStore supports the creation of indexes over collections (e.g., set, bag, and list). While GemStone and O_2 indexes use a B-Tree structure, ObjectStore provides both B-Tree and hash-based indexes.

The data model of ObjectStore is based directly on C++ with some extensions, most notably, support for collection types. The data model is divided between primitive types (e.g. int, float, char) and structured types (e.g. array, struct, class). The data definition and data manipulation language is C++ with extensions including instantiating, manipulating and expressing queries over collections. In contrast to GemStone and O_2, ObjectStore does not store schemas or methods within the database. Instead, schemas and methods are coded using C++ and stored in C++ header files. Moreover, databases are not defined by a particular fixed schema, but incrementally by the schemas of all applications that access the database. ObjectStore provides a general-purpose database browser that can examine data and perform rudimentary value-based queries. However, the browser cannot execute methods.

For the most part, ObjectStore also meets or exceeds the minimal capabilities required by our architecture. ObjectStore is quite different from GemStone and O_2 both functionally and architecturally. ObjectStore is used primarily as a back-end server to provide persistence for C++ application data. The vast majority of processing in an ObjectStore application takes place on the client side. Fortunately, our architecture imposes no restrictions on the architecture of the underlying OODBMS.

Communicating with external data sources from ObjectStore via BSD sockets is also straightforward. ObjectStore methods are coded in C++ and compiled and linked with standard libraries, such as BSD sockets, into each application that accesses ObjectStore.

As one might expect, the division of ObjectStore's data model into primitive and structure types presents the same problem with detecting null attributes for primitive types that we noted for O_2. Fortunately, the problem can be addressed in two ways with ObjectStore. The first solution is similar to the O_2 approach of wrapping primitive types in classes. However, ObjectStore classes cannot be based simply on a primitive type as in O_2, but instead must contain some typed attribute:

OODBMS Fragment 5.5 ObjectStore Primitive Type Wrapper Class Definition

```
class Integer {
   public:
   int value;
};
```

This sort of encapsulation results in a rather clumsy syntax for accessing the value of a wrapped primitive type (e.g., k–> $value = i$–> $value + j$–> $value$).

However, the ObjectStore data model permits typical C++ pointers in class definitions. So a better solution is to simply use pointers to primitive types in a class definition:

OODBMS Fragment 5.6 ObjectStore Class Definition

```
class Crystal : public Solid {
   public:
   int *spaceGroupNumber;
   char *spaceGroup;
   UnitCell *unitCell, *reducedUnitCell;
};
```

This solution still requires care to avoid confusing a pointer to a primitive type with a value of that type in the development of methods (e.g., $*k = *i + *j$).

ObjectStore's explicit persistence is a very different paradigm from the persistence-by-reachability used by both GemStone and O_2. For our purposes, however, the persistence model is not critical. In fact, explicit persistence may be quite beneficial since it prevents the creation of garbage within the OODBMS as data from external sources passes through

the OOHDB. Where we must be careful to reuse previously allocated space by overwriting old data in our GemStone implementation (Section 4.3.2), we can simply use transient objects in ObjectStore and avoid the problem altogether. Of course explicit persistence has its own set of problems, not the least of which is the difficulty of identifying garbage data that was not properly deleted explicitly.

Transient objects also provide a solution to the problem of shared sockets and DEO caches. While these transient objects were created within the database for O_2, this is unnecessary for ObjectStore. Because all method code is compiled and linked into each ObjectStore application, these transient objects can simply be application-global C++ objects. However, as was the case with O_2, if we wish the caches to persist between sessions, some mechanism for making them persist between sessions will be required.

We believe that ObjectStore is a suitable OODBMS for implementing our OOHDB architecture. An ObjectStore OOHDB would require a few changes to the approaches used in GemStone, again, most notably in handling null values. Like O_2, ObjectStore does not provide a wealth of pre-defined classes for building elements of our OOHDB infrastructure. However, since ObjectStore is based directly on C++, add-on class libraries for C++ can be used to supply an ObjectStore database with helper classes. The hash-based indexes provided by ObjectStore might be useful for providing the sort of optimized access to *Crystals* based on chemistry that the *CrystalByElement* alternative access structure provides in our GemStone OOHDB (Section 4.5.5).

5.4 Summary

We have briefly examined two different commercial OODBMS systems and believe that either would be a suitable tool for implementing our OOHDB architecture.

The greatest recurring obstacle to implementation is the mixed type system that divides objects between primitive types and structured types. This division makes it impossible to distinguish null values among primitive types. This null-value problem is an unfortunate consequence of the prevalence of C++-influenced type systems among commercial OODBMSs. Our general solution is to restrict the use of primitive types and use

primitive types wrapped in classes or pointers to primitive types instead. While these solutions impose a coding discipline on the implementors and users of the OOHDB, the solutions effectively provide a notion of null values for primitive types. It is our opinion that many OODBMSs have unfortunately taken a step backwards in data modeling due to the lack of a well-defined and consistent approach to handling missing or null data. However, we expect this issue to be addressed more completely as OODBMSs mature.

Our experience in porting an early GemStone prototype of our OOHDB for materials science to O_2 suggests that the architecture is easy to port and we believe that many popular commercial OODBMSs provide the minimal support required to implement our architecture, giving our approach wide applicability.

Chapter 6

Conclusion

We began our research with the goal of developing an easy-to-implement OOHDB providing access to lightly-managed databases. Our research intended to bridge the gap between so-called metadata approaches that provide a query interface to data sets via a metadata database but fail to integrate heterogeneous data sets, and current HDB approaches that provide a homogeneous view of heterogeneous data but fail to integrate data that is not stored in a powerful DBMS. The result has been the development of a domain- and OODBMS-independent OOHDB architecture providing optimizable access to a variety of data sources including lightly-managed databases.

Our architecture can be easily implemented by database developers faced with integrating heterogeneous data to provide a robust and powerful OOHDB. The architecture is easy to implement because it can be constructed using a commercial OODBMS as the implementation tool. With the advent of object-oriented database technology combining expressive data models with computationally complete behavior, it has become possible to use the database as something more than a passive repository for data. The great benefit of using an OODBMS as an implementation tool is that the database features of the OODBMS (e.g., query language, optimization mechanisms, GUI, transaction mechanism) can be used as-is by the OOHDB. Thus, implementing an OOHDB becomes largely a matter of extending the behavior of the OODBMS to behave like an OOHDB rather than developing all the database functionality for an OOHDB from scratch. The Database Encapsulation and External Data Source Layers of our materials science OOHDB were implemented in approximately 800 lines of commented GemStone SmalltalkDB code and 335 lines of commented C code for the general-purpose BSD socket-based servers.

Users accessing our materials science OOHDB are left with the impression that they have merely been using a typical GemStone database—the heterogeneous aspects being completely hidden.

We believe that the architecture has wide applicability. There are many environments where accessing lightly-managed heterogeneous data is a major concern. In fact, the explosion in automated data generation and collection in science (e.g., Earth Observing Station, Atmospheric Radiation Modeling, mapping the human genome) demands new approaches to integrating large volumes of data that are not stored in traditional database management systems. While traditional metadata approaches are currently in wide use, our work demonstrates that object-oriented database technology provides a solution that can enhance the transparency of heterogeneous data access and provide greater database functionality as well.

6.1 Lessons Learned

Taking our architecture from a concept to an implementation taught us a number of lessons that are somewhat tangential to our thesis but worth noting nonetheless.

6.1.1 OODBMSs are Powerful Tools

First and foremost, using a commercial OODBMS as an implementation tool proved very successful. OODBMSs have matured to the point where they are suitable for complex real-world applications. We cannot overemphasize how much the power of the underlying OODBMS simplified the implementation of our OOHDB. GemStone in particular was a pleasure to work with. For someone interested in building an OOHDB using our architecture, we would recommend using the GemStone OODBMS for four reasons. First, of the OODBMSs we have discussed here, we feel it is the most mature, full-featured, and robust. Second, the incremental development model encouraged by the GemStone architecture makes it much easier to develop and test complex systems. To some degree, incremental development is also possible in O_2, but ObjectStore's use of C++ requires a fully-functional system to be developed as a whole before it can be tested and debugged.

Third, as we noted in Chapter 5, GemStone provides a much richer pre-defined set of classes than many OODBMSs. GemStone classes such as *Dictionary* and *SortedCollection* can be used to implement many of the structures useful in building an OOHDB. Finally, a GemStone-based OOHDB can provide enhanced functionality by taking advantage of GemStone-specific features that give the user much greater access to the internal workings of GemStone. We discuss some ideas for enhanced functionality exploiting GemStone's unique capabilities in Section 6.2.

6.1.2 Data Interchange Formatted Files $\neq$ DBMS

Another lesson we learned was that data interchange formats (DIFs), such as the Crystallographic Interchange File, are not queried very efficiently. DIFs are designed to promote the portable exchange of entire data sets and have limited query support. In general, as the structural flexibility of our data sources increased, the efficiency of accessing them to retrieve data decreased. The proliferation of DIF standards and the historical absence of database systems capable of modeling complex data has led to an undesirable situation where some environments have turned to DIF files as a data storage medium [MH94]. We do not believe that storing large data sets as collections of DIF files is a good long-term solution. We hope that the advent of more powerful data modeling paradigms, such as object-oriented data models and databases, will stem this use of DIF files for storage.

6.1.3 Interfacing an Application Program to the OOHDB

Finally, in the course of our research, we worked to interface a Macintosh-based application program for crystallographic simulation, the Desktop Microscopist, to the OOHDB. In the course of connecting the application to the database, we made an interesting discovery. We found it beneficial to connect the two indirectly. That is, rather than having the Desktop Microscopist construct and execute GemStone queries directly, it sends query parameters to a "query-server" that constructs and executes a GemStone query as shown in Figure 6.1.

The Desktop Microscopist passes query parameters in the form of a string containing triples, *(attribute, relationalOperator, value)*, to the "macServer" which in turn uses

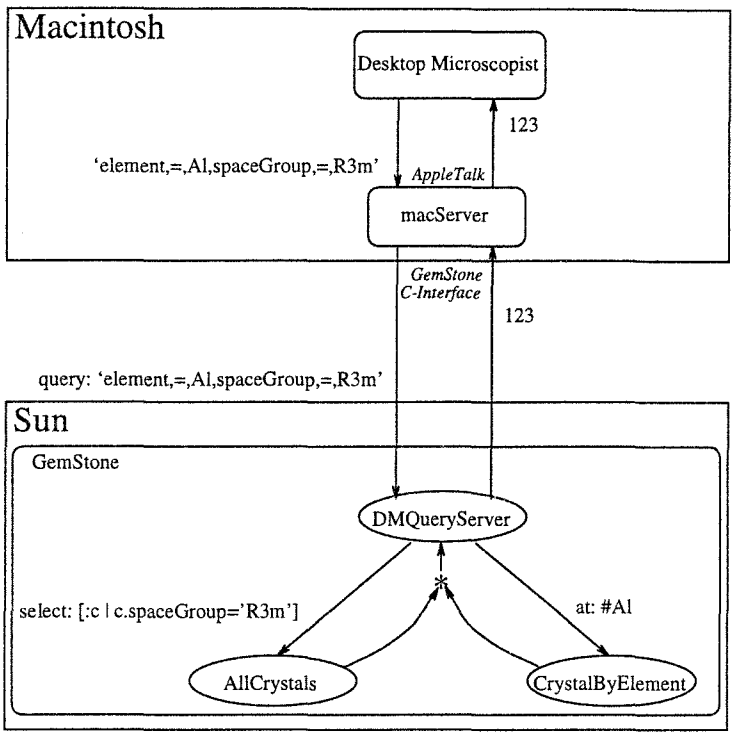


Figure 6.1: Indirect Application Query Interface

the string as the argument to the *query:* method sent to the *DMQueryServer* object in GemStone. The *DMQueryServer* constructs and executes an optimized query, such as the query in Figure 6.1 that uses an index over the *spaceGroup* to select from *AllCrystals* and intersects the result with the subset of *Crystals* containing Aluminum:

SmalltalkDB Fragment 6.1 Optimized Application Query

```
(AllCrystals select: [:c | c.spaceGroup = 'R3m') *
    (CrystalByElement at: #Al)
```

The number of *Crystals* matching the selection criteria is returned by the *DMQueryServer* to the Desktop Microscopist via the macServer. Beyond the *query:* message, the *DMQueryServer* responds to messages that subsequently retrieve one or all of the objects that were selected by a *query:* message, as well as messages that allow the Desktop Microscopist user to store the results of a *query:* in a named collection in GemStone for further reference.

Initially, this indirect query mechanism was the product of necessity since the C-compiler used in the development of the Desktop Microscopist was not compatible with the GemStone C-interface for the Macintosh. So the Desktop Microscopist uses AppleTalk to communicate with the simple macServer on the Macintosh that sends the query parameters on to the *DMQueryServer* object in GemStone using the GemStone C-Interface. However, this indirect query mechanism provides an important benefit that was not foreseen. The benefit is that the *DMQueryServer* object is able to build optimized queries that take advantage of the optimizations and alternative access structures we have built into the OOHDB. Of course, this functionality could have been programmed into the Desktop Microscopist and, in fact, we initially built the query-construction and execution behavior into the macServer running on the Macintosh (compiled with a GemStone C-interface compatible compiler). However, we eventually ported this behavior to the *DMQueryServer* object coded in SmalltalkDB and simplified the macServer. The advantage of having the SmalltalkDB *DMQueryServer* construct and execute queries is that it can be incrementally extended in a matter of minutes as new optimizations and structures are added to the OOHDB without having to modify and recompile a C-coded server. Thus, instead of simultaneously trying to debug a C program, the interface to GemStone, and the queries themselves, we need only debug the queries generated by the SmalltalkDB methods of the *DMQueryServer* in GemStone (using the much friendlier Smalltalk-based debugger built into GemStone). This development and porting effort clearly demonstrated the power and flexibility of extracting common behavior from application programs and embedding them in the OODBMS.

6.2 Future Directions

One of the most obvious topics of future research would be to apply our OOHDB architecture to another data management environment. We would be most interested in an environment with *very* large entities (e.g., satellite images, scientific data sets) that involved a multi-level storage hierarchy (e.g., disk, CD-ROM, robotic tape systems, archive tapes). It would be interesting to analyze the space and time efficiency of the OOHDB

for a particularly challenging environment.

In addition, there are two classes of follow-on research that could continue from our work. First, there are minor enhancements that would refine elements of the architecture or its implementation. These enhancements might take advantage of the features that a particular OODBMS offers for implementation. The second sort of research are the substantial extensions to the architecture to address current deficiencies, improve the architecture substantially, or accommodate new sorts of data sources.

6.2.1 Minor Enhancements

Most of the minor enhancements we envision would occur by exploiting the particular features of the underlying OODBMS. These enhancements would detract a bit from the OODBMS-independent nature of the architecture, but could provide higher levels of performance and transparency for a given OODBMS.

One OODBMS-independent enhancement would be to parse CIFs lazily. There is no reason to parse the entire CIF when extracting a particular piece of data from a CIF. We can imagine extending the CIF class to hold partially-parsed CIF structures that are parsed incrementally as needed. Because logically related data is likely to have close physical proximity within a CIF, we expect that a CIF would probably never be completely parsed as the portion completed would include most of the relevant data. Since CIFs made up such a small portion of our data, however, we did not invest any effort in optimizing the parsing process.

Another relatively OODBMS-independent enhancement would be to use multi-valued logic for representing missing data. We currently rely on a "null" value for attributes to indicate when data should be retrieved from an external data source. The problem with this approach is that the data may be missing from the external data source as well. However, since we have only one way of representing the fact that the data is missing, the architecture naively attempts to retrieve the data from the external data source over and over again each time the null value is encountered. What is needed is another "undefined-Object" (to use Smalltalk parlance) that looks and behaves like null but is distinct from null so that we can detect whether or not a previous attempt to retrieve the data has been

made. While this enhancement is somewhat OODBMS-independent, of the OODBMSs we have examined here, only GemStone provides sufficient access to the internals of the OODBMS to support this sort of strategy.

A GemStone-specific enhancement would be to build support for forwarding messages into the *Behavior* class that controls how methods are executed in GemStone. The idea would be for the basic attribute access method we presented back in Method 4.1 on page 49 to behave exactly like Method 4.2. We believe that simple modifications to the "perform" methods can be used to forward messages on to a DEO when the attribute value is null. However, tinkering with the way GemStone performs methods is not to be undertaken lightly. Such an "enhancement" may have unintended consequences for a GemStone database that manages other sorts of data that are not OOHDB-related.

6.2.2 Major Extensions

We have identified three major architectural extensions that would require substantial follow-on research: handling updates, combining lightly-managed and DBMS external data sources within the OOHDB, and parallelizing external data access.

Handling Updates

Handling updates is a difficult problem for two reasons. First, the fact that the DEO maps data from the heterogeneous representation to the homogeneous schema of the OOHDB means that a corresponding inverse mapping must exist so that changes made within the OOHDB can be written through to the external data source. Where the mappings are trivial and inverse functions easy to develop, we would suggest extending our approach by extending attribute updating methods to forward updates on to DEOs much the same way that attribute accessing methods forward read requests. The DEO would convert and write the updated data back to the external data source.

The second problem with updates is coordinating transactions among the external data sources. Where distributed transactions are supported, the transaction mechanisms of the OODBMS might be used to coordinate a distributed transaction among external data sources. However, distributed transactions are a significant research topic unto themselves.

In the absence of distributed transactions, an alternative strategy is to use some sort of "compensating" transaction in the event of failure. A compensating transaction is an "inverse" transaction that undoes committed portions of a partially-completed distributed transaction [NZ94]. However, compensating transactions are not atomic and leave the external databases temporarily open to inconsistency.

From what we have observed in our work as well as the work of others, the main goal of an HDB is usually to provide integrated *read* access to heterogeneous data. For the most part, modifying data appears to be a task that is best left to the external databases. Thus, it is not clear that providing updates through the OOHDB is a critical need. However, we also observe that in many cases users would like to use heterogeneous data to augment a new application. In that case, our OOHDB provides a unique solution by allowing users to use a common schema for storing their new data within the OOHDB while providing transparent access to external data.

Combining Lightly-Managed and DBMS Data Sources

One of the limitations of our approach is that the static representation we build within the OOHDB limits the applicability of our architecture where data sources are frequently changing. We believe that our architecture can be extended by introducing "proxy collection" objects that represent a whole collection of objects in an external data source. This is the traditional approach taken when building an OOHDB that integrates data from external DBMSs. The proxy collection responds to an OOHDB query by constructing a DBMS-specific sub-query that is passed along to an external DBMS.

We believe that an approach that combines static and proxy collections together can provide uniform and transparent access to a wider variety of external data. In a GemStone OOHDB, we envision creating a new sort of "hybrid" collection class that provides similar behavior to the standard collection classes. A hybrid collection object would hold two very different sorts of objects. One sort of object would be a standard collection containing the static objects we currently use to access lightly-managed data, such as the *AllCrystals* collection. The other sort of object in the hybrid collection would be proxy collection objects that each represent an entire collection of objects stored in an external DBMS.

Queries in the form of *select* messages to the hybrid collection would simply be forwarded to each of the collection objects in the hybrid collection. Standard collection objects would respond by executing and return the results of the *select* message as is done currently. Proxy collection objects would respond by using the selection predicate from the *select* message to construct and execute a sub-query against an external DBMS.

Of course, this sketch of a solution ignores many of the details of implementing this hybrid collection class. True transparency will require that the hybrid collection behave much like a standard collection in every way and it is not clear how difficult that would be to achieve.

In addition, this solution relies on our ability to extend the query behavior of the OODBMS. In fact, of the OODBMSs discussed here, only GemStone provides this sort of access to the semantics of the query language. In most OODBMSs, the query language is a separate and reserved syntax that is built into the OODBMS and not modifiable by the OODBMS user. GemStone is unique in this regard since the query language is comprised of messages to collection classes—messages that can be re-implemented and overridden by user-defined collection subclasses.

Parallelizing External Access

From the results summarized back in the log plot of Figure 4.7 on page 72, it is obvious that accessing external data is costly. What is also obvious is that the OOHDB and CPU are largely idle when accessing external data. This under utilization of the CPU suggests that the throughput of external access could be improved if we accessed multiple data sources in parallel—up to an order-of-magnitude increase in throughput in an optimistic scenario. Of course the increase in throughput would depend on the distribution of external data sources, the sharing of resources (e.g., disks, CPU, network), the relative speed of the different resources, and the nature of the query. For these reasons we did not choose to invest any effort in parallelizing data access.

Nonetheless, parallelizing external data access in an environment with many data sources that are accessed frequently by OOHDB queries could provide substantial performance improvement. One GemStone-specific solution is to use the *RCQueue* (Reduced

Conflict Queue) class that provides inter-transaction communication between GemStone sessions. An RCQueue object is not bound by the transaction semantics of different database sessions and can be safely manipulated by multiple GemStone sessions. Parallelizing access in a GemStone OOHDB would involve spawning separate GemStone sessions for each external data source. DEOs in the user's session would place access requests for data in specific RCQueues for the other GemStone sessions to perform. The user's session would retrieve and process the results returned via a second RCQueue from the other sessions as they became available.

Another solution would be to simply use the current BSD socket interface to external data sources as a sort of request queue. DEOs would flood sockets with all the data requests and then asynchronously retrieve and process the results as they became available. While this may seem to be a solution that could be used in an O_2 or ObjectStore OOHDB, as we noted previously, the query languages for these systems are a reserved syntax. Thus the inability to alter the flow of control within a query means that this solution can not be implemented in OODBMSs with a separate, "closed" query language.

Accessing Non-Traditional Data Sources

In addition to providing access to DBMS data as discussed above, it would also be interesting to explore accessing non-database sorts of data sources from the OOHDB. These data sources might include remote ftp archives, on-line information services, or even real-time sensor data. An interesting research question is how to provide access to dynamic external data that is not bound by the transaction semantics of the OOHDB.

As these suggestions demonstrate, there are a number of ways our OOHDB architecture, and specific implementations of the OOHDB, could be extended to provide both greater functionality, transparency, and higher levels of performance.

Bibliography

[Ald93] Peter Aldhous. Managing the genome data deluge. *Science*, 262(22):502–503, October 1993.

[AMR94] Peter Aiken, Alice Muntz, and Russ Richards. DoD legacy systems: Reverse engineering data requirements. *Communications of the ACM*, 37(5):26–41, May 1994.

[Ber85] G. Bergerhoff. The inorganic crystal structure data base. In Sheldrick et al. [SKG85], pages 85–95.

[Ber91] Elisa Bertino. Integration of heterogeneous data repositories by using object-oriented views. In Kambayashi et al. [KRS91], pages 22–29.

[BHP92] M.W. Bright, A.R. Hurson, and Simin H. Pakzad. A taxonomy and current issues in multidatabase systems. *Computer*, 25(3):50–60, March 1992.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[Cat91] R.G.G. Cattell, editor. *Communications of the ACM—Special Section: Next-Generation Database Systems*, volume 34. Association for Computing Machinery, ACM Press, October 1991.

[Cat93] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Francisco, CA, 1993.

[CD93] E.S. Cordingley and H. Dai. Encapsulation—an issue for legacy systems. *BT Technology Journal*, 11(3):52–64, July 1993.

[CL88] Tim Connors and Peter Lyngback. Providing uniform access to heterogeneous information bases. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 162–173. Springer-Verlag, New York, September 1988.

[CM84] George Copeland and David Maier. Making Smalltalk a database system. In Beatrice Yormark, editor, *Proceedings of Annual Meeting SIGMOD*, volume 14, pages 316–325, Boston, MA, June 1984. Association for Computing Machinery, ACM Press.

[CT91] Bogdan Czejdo and Malcolm Taylor. Integration of database systems using an object-oriented approach. In Kambayashi et al. [KRS91], pages 30–37.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–49, October 1991.

[DSH94] E.G. Dombrowski, W.A. Snyder, and H.M. Heckathorn. Metadata management and the VISTA system. In Nunamaker and Sprague [NS94], pages 418–427.

[EP90] Ahmed K. Elmagarmid and Calton Pu. Guest editors' introduction to the special issue on heterogeneous databases. *ACM Computing Surveys*, 22(3):175–181, September 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[FH94] James C. French and Hans Hinterberger, editors. *Seventh International Working Conference on Scientific and Statistical Database Management*, Charlottesville, VA, September 1994. IEEE Computer Society Press.

[FJP90] James C. French, Anita K. Jones, and John L. Pfaltz. Report of the first invitational NSF workshop on scientific database management (final report). Technical Report 90-21, University of Virginia, Charlottesville, VA, August 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[HAB91] Sydney R. Hall, Frank H. Allen, and I. David Brown. The Crystallographic Information File (CIF): a new standard archive file for crystallography. *Acta Crystallographica*, A47:635–859, November 1991.

[Hal85] Sydney R. Hall. The design, development and implementation of program systems. In Sheldrick et al. [SKG85], pages 147–156.

[HM94] David M. Hansen and David Maier. Using an object-oriented database to encapsulate heterogeneous scientific data sources. In Nunamaker and Sprague [NS94], pages 408–417.

[HMSW92] David Hansen, David Maier, James Stanley, and Jonathan Walpole. An object-oriented heterogeneous database for materials science. *Scientific Programming*, 1(2):115–131, Winter 1992.

[HMZ90] Sandra Heiler, Frank Manola, and Stanley Zdonik. An object-oriented database approach to federated systems. [Online], Available FTP: ftp.gte.com Directory: /pub/dom/reports/ File: HEIL90.ps, 1990.

[HS91] Christy Hightower and Robert Schwarzwalder. A comprehensive look at materials science databases. *Database*, 14(2):42–53, April 1991.

[Int90] International Centre for Diffraction Data. *NBS*AIDS83*, November 1990.

[Jir93] Gregory A. Jirak. Aurora Dataserver: A data management system for visualization applications. XIDAK Inc., July 1993.

[Kar94] Peter D. Karp. Report of the workshop on interconnection of molecular biology databases. Technical Report SRI-AIC-549, Stanford Research Institute International, Menlo Park, CA, August 1994.

[KDN90] M. Kaul, K. Drosten, and E.J. Neuhold. ViewSystem: Integrating heterogeneous information bases by object-oriented views. In *Sixth International Conference on Data Engineering*, pages 2–10, Los Angeles, CA, February 1990. IEEE Computer Society Press.

[Kim95a] Won Kim. Introduction to part 2: Technology for interoperating legacy databases. In *Modern Database Systems: The Object Model, Interoperability, and Beyond* [Kim95b], chapter 25, pages 515–520.

[Kim95b] Won Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Addison-Wesley, New York, 1995.

[KP86] Ted Kaehler and Dave Patterson. *A Taste of Smalltalk.* W.W. Norton & Company Inc., New York, 1986.

[KRS91] Y. Kambayashi, M. Rusinkiewicz, and A. Sheth, editors. *First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991. IEEE Computer Society Press.

[LA86] Witold Litwin and Abdelaziz Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(12):10–18, December 1986.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[LM91] Qing Li and Dennis McLeod. An object-oriented approach to federated databases. In Kambayashi et al. [KRS91], pages 64–70.

[LMR90] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LRV90] Christopher Lécluse, Philippe Richard, and Fernando Velez. O_2, an object-oriented data model. In Zdonik and Maier [ZM90], pages 227–236.

[Man89] Frank Manola. Applications of object-oriented database technology in knowledge-based integrated information systems. In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, Selected Reprint Series, pages 126–134. IEEE Computer Society Press, New York, 1989.

[Mes84] S.V. Meschel. Numeric databases in the sciences. *Online Review*, 8(1):77–103, February 1984.

[MH94] David Maier and David M. Hansen. Bambi meets Godzilla: Object databases for scientific computing. In French and Hinterberger [FH94], pages 176–184.

[MS90] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In Zdonik and Maier [ZM90], pages 167–185.

[NIS92] NIST standard reference data products catalog, 1992.

[NS94] Jay F. Nunamaker and Ralph H. Sprague, editors. *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume III, Maui, Hawaii, January 1994. The University of Hawaii, IEEE Computer Society Press.

[NZ94] Marian H. Nodine and Stanley B. Zdonik. Automating compensation in a multidatabase. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on*

System Sciences, volume II, pages 293–302, Maui, Hawaii, January 1994. The University of Hawaii, IEEE Computer Society Press.

[Obj92] Object Design, Inc., Burlington, MA. *ObjectStore User Guide*, October 1992.

[OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD*, volume 21, pages 403–412, San Diego, CA, June 1992. ACM Press.

[Ram91] Sudha Ram. Heterogeneous distributed database systems. *IEEE Computer*, 24(12):7–10, December 1991.

[RD94] Barbara Rieche and Klaus R. Dittrich. A federated DBMS-based integrated environment for molecular biology. In French and Hinterberger [FH94], pages 118–127.

[RL85] John R. Rumble and David R. Lide. Chemical and spectral databases: A look into the future. *Journal of Chemical Information and Computer Sciences*, 25(3):231–235, 1985.

[Rum89] John R. Rumble. Socioeconomic barriers in computerizing materials data. In Jerry S. Glazman and John R. Rumble, editors, *Computerization and Networking of Materials Data Bases*, pages 217–226. ASTM, Philadelphia, PA, 1989.

[SAD+95] Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, and William Kent. Pegasus: A heterogeneous information management system. In Kim [Kim95b], chapter 32, pages 664–682.

[SBD+81] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W.T. Lin, and Eugene Wong. Multibase – integrating heterogeneous distributed database systems. In *AFIPS Proceedings of the National Computer Conference*, volume 50, pages 487–489, Reston, VA, 1981. American Federation of Information Processing Societies, Inc., AFIPS Press.

[SCGS91] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of data models as canonical models for federated databases. In Arie Segev, editor, *SIGMOD Record—Special Issue: Semantic Issues in Multidatabase Systems*, volume 20, pages 44–48. ACM Press, December 1991.

[Sha93] Ming-Chien Shan. Pegasus architecture and design principles. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22, pages 422–425, Washington, DC, June 1993. Association for Computing Machinery, ACM Press.

[SKG85] G.M. Sheldrick, C. Kruger, and R. Goddard, editors. *Cyrstallographic Computing 3: Data Collection, Structure Determination, Proteins, and Databases.* Oxford University Press, New York, 1985.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SR94] Dong-Guk Shin and Francois Rechenmann. Data and knowledge base issues in genomics. In Lawrence Hunter, editor, *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume V, pages 3–4, Maui, Hawaii, January 1994. The University of Hawaii, IEEE Computer Society Press.

[TSB92] Henry R. Tirri, Jagannathan Srinivasan, and Bharat Bhargava. Integrating distributed data sources using federated objects. In M. Tamer Ozsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Pre-Proceedings of the International Workshop on Distributed Object Management*, pages 292–306, Edmonton, Canada, August 1992.

[WH94] M.H. Williams and J. Hu. Making heterogeneous medical databases interoperable. *Computer Methods and Programs in Biomedicine*, 43(3/4):275–282, June 1994.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[Wil85] Martha E. Williams. Electronic databases. *Science*, 228:445–456, April 1985.

[XID94] XIDAK Inc., Palo Alto, CA. *The Aurora Dataserver Programmer's Guide*, 1994.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Francisco, CA, 1990.

Appendix A

Materials Science Classes

VirtualObject (Object subclass)
- *Class Variables*
  - virtualObjectCaching
- *Class Methods*
  - *Instance Creation*
    - withDEO:withKey:
  - *Accessing*
    - virtualObjectCaching
    - *Updating*
      - virtualObjectCaching:
  - *Instance Variables*
    - deo
    - deoKey
  - *Instance Methods*
    - *Accessing*
      - deo
      - deoKey
    - *Private*
      - deo:
      - deoKey:

Atom (VirtualObject subclass)
- *Class Methods*
  - *Instance Creation*
    - new
    - element:
    - element:electronConfiguration:
    - element:nuclearConfiguration:
    - element:nuclearConfiguration:electronConfiguration:
- *Instance Variables*
  - element
  - electronConfiguration
  - nuclearConfiguration
- *Instance Methods*
  - *Updating*
    - electronConfiguration:
    - element:
    - nuclearConfiguration:
  - *Accessing*
    - electronConfiguration
    - element
    - nuclearConfiguration

asString

DSpacing (VirtualObject subclass)

Instance Variables

- d
- intensity
- h
- k
- l

Instance Methods

Accessing

- d
- intensity
- h
- k
- l

Updating

- d:
- intensity:
- h:
- k:
- l:

ElectronConfiguration (VirtualObject subclass)

Instance Variables

- element
- electronConfiguration
- electronAffinity
- electronNegativity
- orbital
- valence
- atomicRadii
- ionizationPotential

Instance Methods

Updating

- element:
- electronConfiguration:
- electronAffinity:
- electronNegativity:
- orbital:
- valence:
- atomicRadii:
- ionizationPotential:

Accessing

- element
- electronConfiguration
- electronAffinity
- electronNegativity
- orbital
- valence
- atomicRadii
- ionizationPotential

Element (VirtualObject subclass)

Instance Variables

- name
- symbol
- atomicNumber
- atomicMass

discovery
defaultElectronConfiguration
defaultNuclearConfiguration
electronConfigurations
nuclearConfigurations

Instance Methods

Updating

name:
symbol:
atomicNumber:
atomicMass:
discovery:
defaultElectronConfiguration:
defaultNuclearConfiguration:
electronConfigurations:
nuclearConfigurations:

Accessing

name
symbol
atomicNumber
atomicMass
discovery
defaultElectronConfiguration
defaultNuclearConfiguration
electronConfigurations
nuclearConfigurations
asString

Formula (VirtualObject subclass)

Class Methods

Instance Creation

fromString:

Instance Variables

atoms
formula

Instance Methods

Testing

includesAtom:

Comparing

=

Updating

atoms:
formula:

Accessing

atoms
formula
asString

JournalReference (VirtualObject subclass)

Instance Variables

author
coden
page
volume
year

Instance Methods

Updating

author:
coden:
page:

volume:
year:
Accessing
author
coden
page
volume
year

Material (VirtualObject subclass)

Instance Variables
name
phases
journalReference
Instance Methods
Accessing
name
phases
journalReference
Updating
name:
phases:
journalReference:

Molecule (VirtualObject subclass)

Instance Variables
name
formula
empiricalFormula
Instance Methods
Updating
name:
formula:
empiricalFormula:
Accessing
name
formula
empiricalFormula
asString

NuclearConfiguration (VirtualObject subclass)

Instance Variables
element
isotope
abundance
nuclearMagneticMoment
thermalNCrossSection
nuclearSpin
bScatteringLength
coherentCrossSection
totalScatter
Instance Methods
Updating
element:
isotope:
abundance:
nuclearMagneticMoment:
thermalNCrossSection:

nuclearSpin:
bScatteringLength:
coherentCrossSection:
totalScatter:

Accessing

element
isotope
abundance
nuclearMagneticMoment
thermalNCrossSection
nuclearSpin
bScatteringLength
coherentCrossSection
totalScatter

Phase (VirtualObject subclass)

Instance Variables

name
molecule
density
meltingTemperature
boilingTemperature
journalReference

Instance Methods

Updating

name:
molecule:
density:
meltingTemperature:
boilingTemperature:
journalReference:

Accessing

name
molecule
density
meltingTemperature
boilingTemperature
journalReference

Gas (Phase subclass)

Liquid (Phase subclass)

Solid (Phase subclass)

Crystal (Solid subclass)

Instance Variables

unitCell
spaceGroup
spaceGroupNumber
reducedUnitCell
dSpacings
maxdSpacing

Instance Methods

Formatting

forDesktopMicroscopist

Updating

unitCell:
reducedUnitCell:
spaceGroup:
dSpacings:
maxdSpacing:
addToAccessPaths
deleteFromAccessPaths

Accessing

spaceGroup
unitCell
reducedUnitCell
dSpacings
spaceGroupNumber
maxdSpacing

Glass (Solid subclass)

QuasiCrystal (Solid subclass)

UnitCell (VirtualObject subclass)

Instance Variables

a
b
c
alpha
beta
gamma
volume

Instance Methods

Comparing

=

Updating

a:
b:
c:
alpha:
beta:
gamma:
volume:

Accessing

a
b
c
gamma
alpha
beta
volume

Appendix B

Database Examples

B.1 A NBS Crystal/PDF-2 Database Record

Record types, identified by the character in column 80, contain the following information:

1 — Cell Parameters

2 — Cell Parameter Standard Deviations

3 — Space Group, Z, Density

4 — Crystal Data Space Group, Z, Density

5 — File, Class, and Registration Indicators

6 — Compound Name

7 — Chemical Formula

8 — Empirical Formula

9 — Literature Reference

A — Structure type

B — Comments

C — Matrix for Initial Cell—Crystal Data Cell

D — Reduced Cell

E — Crystal Data Cell

F — Pattern Information (1)

G — Pattern Information (2)

H — Extinction Conditions

I — Powder Pattern

J — Update/Revision

+ — Information for Hanawalt and Fink Indexes

***** — Information for Max-d Index

K — Processing History and Entry Termination

```
                                                          M                UP381598X1
                                                       232.28        0.00  P381598X4
 0                                                       2210-63-1         P381598X5
Monophenylbutazone                                                      P  1P381598X6
mofebutazone                                                            C  2P381598X6
C13 H16 N2 O2                                                              P381598X7
C13 H16 N2 O2                                                              P381598X8
OPCOMC          1987 Rose, H., Eli Lilly and Company, Indianapolis,     C1P381598X9
                     Indiana, USA.                                         2P381598X9
Assay 100.1% by N.A.T.                                                     1P381598XB
100.5-102.5 C                                                           MP 2P381598XB
CuKa  1.5418        F Ni     DD         S 114.6                            P381598XF
F                      BB                                                  P381598XG
11.0900100             9.80000  3            6.75000 13                    1P381598XI
5.76000 16             5.45000  5            4.82000 45                    2P381598XI
4.48000 26             4.34000 15            4.10000 23                    3P381598XI
3.97000  3             3.72000 39B           3.65000  3                    4P381598XI
3.41000 17             3.29000 29            3.21000  3                    5P381598XI
3.09000  1             2.97000  8            2.88000  5                    6P381598XI
2.78000  1             2.69000  8B           2.48000  1                    7P381598XI
2.31000  3             2.23000  2            2.17000  3                    8P381598XI
2.05000  1             1.98400  4            1.93200  1                    9P381598XI
B 11.1/X 4.82/5 3.72/4 3.29/3 4.48/3 4.10/2 3.41/2 5.76/2 4.34/2 6.75/1P381598X+
B 11.1/X 9.80/1 6.75/1 5.76/2 5.45/1 4.82/5 4.48/3 4.34/2 4.10/2 3.97/1P381598X*
01/31/87  03/27/87 tk  88/ 2/24  0  0 10 86/03/05 P-22564                 P381598XK
```

B.2 A Desktop Microscopist File

```
Secondary Source =  0
Lattice Type =  6
Space Group =  193
a =  7.5450
b =  7.5450
c =  7.5450
alpha =  90.0000
beta =  90.0000
gamma =  120.0000
Volume =  123.9901
Reduced a =  0.0000
Reduced b =  0.0000
Reduced c =  0.0000
Reduced alpha =  0.0000
Reduced beta =  0.0000
Reduced gamma =  0.0000
Reduced Volume =  123.9901
JCPDF File# =  0
JCPDF Card# =  0
K W 3 O 9



Year Discovered =  1800
Element List, (#) =  3
  =  18
  =  73
  =  7
Atom Positions (#) =  26
19 1 0.0000 0.0000 0.2500
19 1 0.0000 0.0000 0.7500
74 7 0.4800 0.0000 0.2500
74 7 0.0000 0.4800 0.2500
74 7 0.5200 0.5200 0.2500
74 7 0.5200 0.0000 0.7500
74 7 0.0000 0.5200 0.7500
74 7 0.4800 0.4800 0.7500
8 6 0.5000 0.0000 0.0000
8 6 0.0000 0.5000 0.0000
```

```
8 6 0.5000 0.5000 0.0000
8 6 0.5000 0.0000 0.5000
8 6 0.0000 0.5000 0.5000
8 6 0.5000 0.5000 0.5000
8 10 0.4550 0.6670 0.2500
8 10 0.3330 0.7880 0.2500
8 10 0.2120 0.5450 0.2500
8 10 0.5450 0.3330 0.7500
8 10 0.6670 0.2120 0.7500
8 10 0.7880 0.4550 0.7500
8 10 0.6670 0.4550 0.2500
8 10 0.7880 0.3330 0.2500
8 10 0.5450 0.2120 0.2500
8 10 0.3330 0.5450 0.7500
8 10 0.2120 0.6670 0.7500
8 10 0.4550 0.7880 0.7500
Wyckoff Positions (#) =  4
a2 1 19 0.0000 0.0000 0.0000
g6 7 74 0.4800 0.0000 0.2500
f6 6 8 0.2200 0.0000 0.2500
j12 10 8 0.4550 0.6670 0.8990
#User Defined Atoms (#) =  0
Listing of Dspacings + Intensities
0 1 0 6.5342 23.9844
1 0 0 6.5342 23.9844
1 1 0 3.7725 2.8415
0 0 2 3.7725 73.9098
1 1 1 3.3742 10.8603
0 2 0 3.2671 62.0676
2 0 0 3.2671 62.0676
0 1 2 3.2671 12.8614
1 0 2 3.2671 12.8614
1 1 2 2.6676 4.0787
2 0 2 2.4697 33.5595
0 2 2 2.4697 33.5595
1 2 0 2.4697 9.4727
2 1 0 2.4697 9.4727
2 1 1 2.3471 8.8404
1 2 1 2.3471 8.8404
0 3 0 2.1781 8.9813
3 0 0 2.1781 8.9813
1 1 3 2.0926 7.7465
```

```
2 1 2 2.0663 3.3598
1 2 2 2.0663 3.3598
0 3 2 1.8863 3.4075
3 0 2 1.8863 3.4075
2 2 0 1.8863 37.5820
2 2 1 1.8299 8.5205
3 1 0 1.8123 6.2491
1 3 0 1.8123 6.2491
1 3 1 1.7621 6.7877
1 2 3 1.7621 6.9849
3 1 1 1.7621 6.7877
Elastic Constants
1.2900 1.0700 1.0700 0.0000 0.0000 0.0000
1.2900 1.0700 0.0000 0.0000 0.0000
1.2910 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
0.8200 0.0000
0.1100
Physical Parameters (#) =  0
Thermodynamic Parameters (#) =  0
Crystal Faces (#) =  6
1 0 0 13.0683
-1 0 0 13.0683
0 1 0 13.0683
0 -1 0 13.0683
0 0 1 15.0900
0 0 -1 15.0900
```

B.3 A CACHe File

```
molstruct68_Dec_11_1991_10:20:01 new 0x0100
Written by Molecule Editor on Thu Aug 13 14:18:54 1992
Using Data Dictionary        11/4/91  2:59 PM
Version 2
local_transform
0.091256 0.000000 0.000000 0.000000
0.000000 0.091256 0.000000 0.000000
0.000000 0.000000 0.091256 0.000000
-0.061712 -0.061712 -0.061712 1.000000
object_class connector
property dflag MoleculeEditor noUnit 0 1 HEX
property objCls1 MoleculeEditor noUnit 0 1 NAME
property objID1 MoleculeEditor noUnit 0 1 INTEGER
property objCls2 MoleculeEditor noUnit 0 1 NAME
property objID2 MoleculeEditor noUnit 0 1 INTEGER
ID dflag objCls1 objID1 objCls2 objID2
 1   0x0    atom      1 crystal      1
 2   0x0    atom      2 crystal      1
 3  0xa0    atom      2    bond      1
 4  0xa0    atom      1    bond      1
property_flags:
object_class crystal
property a MoleculeEditor angstrom 4 1 FLOAT
property b MoleculeEditor angstrom 4 1 FLOAT
property c MoleculeEditor angstrom 4 1 FLOAT
property alpha MoleculeEditor degree 5 1 FLOAT
property beta MoleculeEditor degree 5 1 FLOAT
property gamma MoleculeEditor degree 5 1 FLOAT
property rflag MoleculeEditor noUnit 0 1 HEX
property CellMultiplier MoleculeEditor noUnit 3 3 FLOAT
property Space-Group MoleculeEditor noUnit 0 32 STRING
property Descriptor MoleculeEditor noUnit 0 80 STRING
ID a      b      c      alpha    beta     gamma    rflag  CellMultiplier
 1 5.4100 5.4100 5.4100 90.00000 90.00000 90.00000 0x7043 1.000 1.000 1.000
property_flags:
ID Space-Group Descriptor
 1       Fm-3m       none
property_flags:
object_class atom
property rflag MoleculeEditor noUnit 0 1 HEX
```

```
property sym MoleculeEditor noUnit 0 2 STRING
property xyz_coordinates MoleculeEditor angstrom 7 3 FLOAT
property anum MoleculeEditor unit 0 1 INTEGER
property chrg MoleculeEditor charge_au 0 1 INTEGER
property conf MoleculeEditor noUnit 0 1 NAME
property Label MoleculeEditor noUnit 0 3 STRING
ID rflag  sym xyz_coordinates               anum chrg conf  Label
 1 0x5052   O 1.3525000 1.3525000 1.3525000    8    0   sp3      O
 2 0x5052  Ce 0.0000000 0.0000000 0.0000000   58    0 d2sp3     Ce
property_flags:
object_class bond
property type MoleculeEditor noUnit 0 1 NAME
property rflag MoleculeEditor noUnit 0 1 HEX
ID type   rflag
 1 single 0xf045
property_flags:
```

B.4 A CIF File

```
##############################################################################
###    CIF submission form for molecular structure report (Acta Cryst. C)  ###
###                                                Version: 7 October 1991  ###
##############################################################################

# This is an electronic "form" for submitting a structural paper to Acta Cryst.
# Section C as a Crystallographic Information File. Full details of the format
# of such files are given in the paper "The Crystallographic Information File
# (CIF): a New Standard Archive File for Crystallography" by S. R. Hall, F. H.
# Allen and I. D. Brown [Acta Cryst. (1991), A47, 655-685]. An example of a
# completed CIF form may be obtained by sending the one-line request
#      'send example.cif'
# to sendcif@iucr.ac.uk. Queries or requests for further information should be
# directed to teched@iucr.ac.uk.
#
# Note that all fields should be numeric or character type EXCEPT those which
# are flagged as 'text' - free-form text of any length may be included in
# these latter fields provided the text block begins and ends with a semicolon
# as the first character of a new line. . Note also that the query marks
# '?' are significant as placeholders, and should not be deleted where a data
# item is not given, UNLESS the accompanying data name is also deleted.
# Lines should not exceed 80 characters in length.

#==============================================================================
data_global
#==============================================================================

# 1. SUBMISSION DETAILS

_publ_contact_author            # Name and address of author for correspondence
;
      Dr Anthony C. Willis
      Research School of Chemistry
      Australian National University
      GPO Box 4
      Canberra, A.C.T.
      Australia    2601
;
_publ_contact_author_phone        '616 249 4109'
```

```
_publ_contact_author_fax          '616 249 0750'
_publ_contact_author_email        willis@RSC3.anu.oz.au

_publ_requested_journal           'Acta Crystallographica C'
_publ_requested_coeditor_name     ?

_publ_contact_letter
;
     Please consider this CIF submission for publication as a New Structure
     paper in Acta Crystallographica Section C.  The figures, chemical
     structure diagram (scheme), Transfer of Copyright Agreement form and
     structure factors will be sent on receipt of your acknowledgement letter.
;

#==============================================================================

# 2. PROCESSING SUMMARY (IUCr Office Use Only)

_journal_date_recd_electronic     ?

_journal_date_to_coeditor         ?
_journal_date_from_coeditor       ?
_journal_date_accepted            ?

_journal_date_printers_first      ?
_journal_date_printers_final      ?
_journal_date_proofs_out          ?
_journal_date_proofs_in           ?

_journal_coeditor_name            ?
_journal_coeditor_code            ?
_journal_coeditor_notes
; ?
;

_journal_techeditor_code          ?
_journal_techeditor_notes
; ?
;

_journal_coden_ASTM               ?
_journal_name_full                ?
```

```
_journal_year                    ?
_journal_volume                  ?
_journal_issue                   ?
_journal_page_first              ?
_journal_page_last               ?

_journal_suppl_publ_number       ?
_journal_suppl_publ_pages        ?

#=============================================================================

# 3. TITLE AND AUTHOR LIST

_publ_section_title
;
trans-3-Benzoyl-2-(tert-butyl)-4-(iso-butyl)-1,3-oxazolidin-5-one
;

# The loop structure below should contain the names and addresses of all
# authors, in the required order of publication. Repeat as necessary.

loop_
_publ_author_name
_publ_author_address
'Willis, Anthony C.'            #<--'Last name, first name'
;     Research School of Chemistry
      Australian National University
      GPO Box 4
      Canberra, A.C.T.
      Australia    2601
;
     'Beckwith, Athelstan L.J.'
;     Research School of Chemistry
      Australian National University
      GPO Box 4
      Canberra, A.C.T.
      Australia    2601
;
     'Tozer, Matthew J.'
;     Research School of Chemistry
      Australian National University
      GPO Box 4
```

```
      Canberra, A.C.T.
      Australia    2601
;
```

—Data omitted for brevity—

```
#=============================================================================

# 5. CHEMICAL DATA

_chemical_name_systematic
;
trans-3-Benzoyl-2-(tert-butyl)-4-(iso-butyl)-1,3-oxazolidin-5-one
;
_chemical_name_common                 ?
_chemical_formula_moiety              'C18 H25 N O3'
_chemical_formula_structural          ?
_chemical_formula_analytical          ?
_chemical_formula_sum                 'C18 H25 N O3'
_chemical_formula_weight              303.40
_chemical_melting_point               ?
_chemical_compound_source             ?

loop_
     _atom_type_symbol
     _atom_type_description
     _atom_type_scat_dispersion_real
     _atom_type_scat_dispersion_imag
     _atom_type_scat_source
  C  ?  .017  .009  International_Tables_Vol_IV_Table_2.3.1
  H  ?  0  0  International_Tables_Vol_IV_Table_2.3.1
  O  ?  .047  .032  International_Tables_Vol_IV_Table_2.3.1
  N  ?  .029  .018  International_Tables_Vol_IV_Table_2.3.1

#=============================================================================

# 6. CRYSTAL DATA

_symmetry_cell_setting                orthorhombic
_symmetry_space_group_name_H-M        'P 21 21 21'
_symmetry_space_group_name_Hall       'P 2ac 2ab'
loop_
        _symmetry_equiv_pos_as_xyz
```

```
        'x,y,z'
        1/2-x,-y,1/2+z
        1/2+x,1/2-y,-z
        -x,1/2+y,1/2-z

_cell_length_a                    5.959(1)
_cell_length_b                    14.956(1)
_cell_length_c                    19.737(3)
_cell_angle_alpha                 90.0
_cell_angle_beta                  90.0
_cell_angle_gamma                 90.0
_cell_volume                      1759.0(3)
_cell_formula_units_Z             4
_cell_measurement_temperature     293
_cell_measurement_reflns_used     25
_cell_measurement_theta_min       25
_cell_measurement_theta_max       31
_cell_special_details
; ?
;

_exptl_crystal_description        prism
_exptl_crystal_colour             colourless
_exptl_crystal_size_max           0.32
_exptl_crystal_size_mid           0.27
_exptl_crystal_size_min           0.10
_exptl_crystal_density_meas       1.146
_exptl_crystal_density_diffrn     ?
_exptl_crystal_density_method     ?
_exptl_crystal_F_000              656
_exptl_absorpt_coefficient_mu     0.59
_exptl_absorpt_correction_type    'shelx76 gaussian'
_exptl_absorpt_correction_T_min   .933
_exptl_absorpt_correction_T_max   .824

#===============================================================================
```

—Data omitted for brevity—

Appendix C

Database Encapsulator & Cache Classes

C.1 Database Encapsulator Classes

DatabaseEncapsulator (Object subclass)

CACheFileReducedUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

a:
b:
c:
alpha:
beta:
gamma:
volume:

CACheFileUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

a:
b:
c:
alpha:
beta:
gamma:
volume:

CIFReducedUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

volume:
a:
b:
c:
alpha:
beta:
gamma:

CIFUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

volume:
a:
alpha:
b:
beta:
c:
gamma:

DMFileReducedUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

a:
alpha:
b:
beta:
c:
gamma:
volume:

DMFileUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

b:
a:
c:
alpha:
beta:
gamma:
volume:

DatabaseFile (DatabaseEncapsulator subclass)

Class Methods

Private

getData:
getFrom:line:
getFrom:fromPrefix:toSuffix:datatype:

CACheFile (DatabaseFile subclass)

Class Methods

Accessing

spaceGroupString:

Private

cache
cacheKey
getFrom:atKeyword:datatype:
initializeAccess

External Interface

unitCell:
reducedUnitCell:
journalReference:
chemicalFormula:
spaceGroup:
author:
coden:
page:

year:
volume:
spaceGroupNumber:
dSpacings:
maxdSpacing:

CIF (DatabaseFile subclass)
Class Methods
Accessing
author:
year:
volume:
page:
coden:
molecularWeight:
chemicalFormula:
Private
arrayOfStringFromCIF:
cache
cacheKey:
cache:
cacheKey
getData:
getFrom:at:in:dataType:
flatten:
initializeAccess
External Interface
journalReference:
unitCell:
reducedUnitCell:
dSpacings:
maxdSpacing:

DMFile (DatabaseFile subclass)
Class Methods
Accessing
chemicalFormula:
name:
spaceGroupNumber:
maxdSpacing:
dSpacings:
Private
cache
cacheKey
initializeAccess
External Interface
journalReference:
author:
coden:
year:
volume:
page:
unitCell:
reducedUnitCell:

NBSDatabaseRecord (DatabaseEncapsulator subclass)
Class Variables
codeDictionary
Class Methods

Accessing
- alloyFlag:
- authorsAspectEditorial:
- authorsCalculatedDensity:
- authorsMeasuredDensity:
- authorsSpaceGroup:
- authorsSpaceGroupEditorial:
- authorsSpaceGroupNumber:
- authorsZ:
- authorsZEditorial:
- cdReferenceCode:
- casRegistryNumber:
- cdApproximatedDensity:
- cdAspectEditorial:
- cdCalculatedDensity:
- cdSpaceGroup:
- cdSpaceGroupEditorial:
- cdSpaceGroupNumber:
- cdZ:
- cdZEditorial:
- chemicalFormula:
- chemicalFormulaApproximation:
- chemicalFormulaIndex:
- comments:
- compoundClasses:
- compoundName:
- compoundNameIndex:
- crystalSystem:
- databaseFlag:
- empiricalFormula:
- empiricalFormulaApproximation:
- empiricalFormulaEditorial:
- inorganicFlag:
- mineralFlag:
- molecularWeight:
- molecularWeightEditorial:
- organicFlag:
- structurePearsonEditorial:
- structurePearsonSymbol:
- structureType:
- dSpacings:
- maxdSpacing:
- referenceIntensityRatio:
- referenceCode:
- journalVolume:
- journalCoden:
- journalAuthor:
- journalYear:
- journalPage:

Initialization
- getKeysStartingAt:

Private
- codeDictionary
- getData:
- longString:lineChar:from:to:
- getFrom:at:from:to:dataType:
- getdSpacings:onlyMax:
- buildCrystalsFrom:to:

External Interface
- spaceGroup:
- spaceGroupNumber:
- journalReference:
- page:

author:
coden:
volume:
year:

CrystalRecord (NBSDatabaseRecord subclass)

Class Methods

Private

cache
cacheKey
socket
initializeAccess

External Interface

reducedUnitCell:
unitCell:

PDFRecord (NBSDatabaseRecord subclass)

Class Methods

Private

cache
cacheKey
socket
initializeAccess

External Interface

reducedUnitCell:
unitCell:

CrystalReducedUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

a:
alpha:
b:
beta:
c:
gamma:
metricSymmetryCode:
reducedFormNumber:
volume:

CrystalUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

aStdDev:
alphaStdDev:
authorsA:
authorsAlpha:
authorsB:
authorsBeta:
authorsC:
authorsGamma:
authorsVolume:
averageErrorinAxialLengths:
bStdDev:
betaStdDev:
cStdDev:
cameraDiameter:
cdA:

cdAlpha:
cdB:
cdBeta:
cdC:
cdDeterminantOfTranformMatrix:
cdFirstDeterminativeRatio:
cdGamma:
cdSecondDeterminativeRatio:
cdTransformMatrix:
cdVolume:
cellEditorial:
errorEditorial:
filter:
filterCode:
gammaStdDev:
instrumentCode:
leastSquaresIndicator:
lowerLimit:
pdfEditorialLeastSquares:
pdfEditorialRhombohedral:
qualityIndex:
radiationOfStudy:
radiationUsed:
sourceOfUnitCellData:
spacingStandard:
standardCode:
structure:
truncationFlag:
wavelength:

External Interface

a:
alpha:
b:
beta:
c:
gamma:
volume:

PDFReducedUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

a:
alpha:
b:
beta:
c:
gamma:
metricSymmetryCode:
reducedFormNumber:
volume:

PDFUnitCell (DatabaseEncapsulator subclass)

Class Methods

Accessing

aStdDev:
alphaStdDev:
authorsA:
authorsAlpha:
authorsB:
authorsBeta:

authorsC:
authorsGamma:
authorsVolume:
averageErrorinAxialLengths:
bStdDev:
betaStdDev:
cStdDev:
cameraDiameter:
cdA:
cdAlpha:
cdB:
cdBeta:
cdC:
cdDeterminantOfTranformMatrix:
cdFirstDeterminativeRatio:
cdGamma:
cdSecondDeterminativeRatio:
cdTransformMatrix:
cdVolume:
cellEditorial:
errorEditorial:
filter:
filterCode:
gammaStdDev:
instrumentCode:
leastSquaresIndicator:
lowerLimit:
pdfEditorialLeastSquares:
pdfEditorialRhombohedral:
qualityIndex:
radiationOfStudy:
radiationUsed:
sourceOfUnitCellData:
spacingStandard:
standardCode:
structure:
truncationFlag:
wavelength:

External Interface

a:
alpha:
b:
beta:
c:
gamma:
volume:

CIFDictionary (SymbolDictionary subclass)

Class Methods

Instance Creation

fromStringArray:

Instance Methods

Searching

findDataAt:

CIFLoop (CIFDictionary subclass)

Class Methods

Instance Creation

fromStringArray:

Instance Methods

Accessing
dataName

CIFString (String subclass)

C.2 Cache Class

Cache (Object subclass)
Class Methods
Instance Creation
new:someKeys:data:sortBlock:
Instance Variables
keys
data
lineToReplace
Instance Methods
Accessing
keys
data
lineToReplace
at:ifAbsent:
Updating
keys:
data:
lineToReplace:
incrementLineToReplace

Appendix D

Socket-Based Servers

C Program D.1 Generic Byte-Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>

/*
 *      This program is a general-purpose byte server.  It accepts two
 *      parameters, a TCP port number, and a filename.
 *
 *      The program has two components.
 *
 *      The first component is the parent.  The parent listens for
 *      new service requests and forks a new copy of itself for each
 *      request.
 *
 *      The second component is the child.  The child accepts read
 *      requests of the form:
 *              byte-offset   number-of-bytes
 *      The specified number of bytes are read from the file and
 *      returned to the requester.  When the requester closes the
 *      socket, the child exits.
 */

FILE *file;

main(argc, argv, argp)
    char **argv, **argp;
    int argc;
{
```

```
int serverSock, clientSock;
struct sockaddr_in server;
extern void serveClient();

/* There should be a port number and filename parameter. */
if (argc != 3)
{
    perror("no filename and/or port number");
    exit(1);
}

/* Open the file */
if (!(file = fopen(argv[2], "r")))
{
    perror("opening file");
    exit(1);
}

/* create a socket */
serverSock = socket(AF_INET, SOCK_STREAM, 0);
if (serverSock < 0)
{
    perror("opening stream socket");
    exit(1);
}

/* Name socket using wildcards, and bind to the port. */
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = atoi(argv[1]);
if (bind(serverSock, (struct sockaddr *)&server, sizeof server) < 0)
{
    perror("binding stream socket");
    exit(1);
}

/*
 *  The parent listens to the socket forever, waiting for connect
 *  requests.  Once a connection is requested, a child is forked to
 *  service the request.
 */
listen(serverSock, 100);
for(;;)
{
    /* Wait for a connection request, fork a child. */
    if ((clientSock = accept(serverSock, (struct sockaddr *)0, (int *)0)) < 0)
```

```
        {
            perror("connecting to client socket");
            continue;
        }

        /*
         * If we're the child, not the parent, then service the
         * request.
         */
        if (fork() == 0)
            serveClient(clientSock);
        else
            /* Close the server socket and wait for new request */
            close(serverSock);

    } /* end for */

} /* end Byteserver */



void serveClient(msgsock)
int msgsock;
{
    long offset, numbytes, last_offset = 0, last_numbytes = 0;
    char inbuf[1024], outbuf[8192];
    int rval;
    int i,k;

    /*
     * Continue servicing requests from this client until they close
     * the socket.
     */
    do
    {
       /* Clear the input buffer */
       bzero(inbuf, sizeof inbuf);

       /* Read a message */
       if ((rval = read(msgsock, inbuf, 1024)) < 0)
           perror("reading stream message");

       /*
        *  If the message wasn't a "close" message then read
        *  data from the file and send the data to the client
        */
```

```
        if (rval)
        {
            /* Read the offset and # of bytes from the message */
            sscanf(inbuf,"%d %d", &offset, &numbytes);

            /*
             * If the offset was negative or the number of bytes <=0
             * return an empty message, the request was bad.
             */
            if (offset < 0 || numbytes <= 0)
                write(msgsock, inbuf, 0);
            else
            {
                /*
                 * If this is a repeat request, no need to read the data,
                 * just return the un-changed outbuf again.
                 */
                if (offset == last_offset && numbytes == last_numbytes)
                    write(msgsock, outbuf, numbytes);
                else
                    /* Read the data from the file */
                    if (fseek(file, offset, 0))
                        write(msgsock, outbuf, 0);
                    else
                        /* If the read failed, return an empty message */
                        if (!fread(outbuf, 1, numbytes , file))
                            write(msgsock, outbuf, 0);
                        else
                            /* Send the data to the requester */
                            write(msgsock, outbuf, numbytes);

            } /* end else */

            /* Keep track of the last request */
            last_offset = offset;
            last_numbytes = numbytes;

        } /* end if */

    } while (rval != 0);

    close(msgsock);
    exit(0);

} /* end serveClient*/
```

C Program D.2 Generic File-Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>

/*
 * This program is a general-purpose file server.  It accepts one
 * parameter, a TCP port number.
 *
 * The program has two components.
 *
 * The first component is the parent.  The parent listens for
 * new service requests and forks a new copy of itself for each
 * request.
 *
 * The second component is the child.  The child accepts read
 * requests of the form:
 *      file_pathname
 * The specified file is opened and its contents read and
 * sent to the requester. When the requester closes the
 * socket, the child exits.
 */

main(argc, argv, argp)
   char **argv, **argp;
   int argc;
{
   int clientSock, serverSock;
   struct sockaddr_in server;
   extern void serveClient();

   /* There should be a port number parameter. */
   if (argc != 2) {
      perror("no port number");
      exit(1);
   }

   /* create a socket */
   serverSock = socket(AF_INET, SOCK_STREAM, 0);
   if (serverSock < 0) {
      perror("opening stream socket");
      exit(1);
   }
```

```
    /* Name socket using wildcards, and bind to the port. */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = atoi(argv[1]);
    if (bind(serverSock, (struct sockaddr *)&server, sizeof server) < 0) {
       perror("binding stream socket");
       exit(1);
    }

    /*
     *  The parent listens to the socket forever, waiting for connect
     *  requests.  Once a connection is requested, a child is forked to
     *  service the request.
     */
    listen(serverSock, 100);
    for(;;) {
       /* Wait for a connection request, fork a child. */
       if ((clientSock = accept(serverSock, (struct sockaddr *)0, (int *)0)) < 0) {
             perror("connecting to client socket");
             continue;
       }

       /*
        * If we're the child, not the parent, then service the
        * request.
        */
       if (fork() == 0)
          serveClient(clientSock);
       else
          /* Close the server socket and wait for new request */
          close(serverSock);

    } /* end for */

} /* end Fileserver */



void serveClient(msgsock)
    int msgsock;
{
    FILE *file;
    long offset, numbytes, last_offset = 0, last_numbytes = 0;
    char inbuf[1024], outbuf[8192];
    int rval;
```

```
    int i,k;

    /*
     * Continue servicing requests from this client until they close
     * the socket.
     */
    do {
       /* Clear the input buffer */
       bzero(inbuf, sizeof inbuf);

       /* Read a message */
       if ((rval = read(msgsock, inbuf, 1024)) < 0)
          perror("reading stream message");

       /*
        *  If the message wasn't a "close" message then read
        *  data from the file and send the file to the client
        */
       if (rval) {
          /* Open the file using the pathname from the message */
          file = fopen(inbuf, "r");

          /*
           * If the file open succeeded, read and send the file in
           * manageable chunks until EOF.
           */
          if (file)
             while (!feof(file)) {
                numbytes = fread(outbuf, 1, sizeof(outbuf), file);
                write(msgsock, outbuf, numbytes);
             } /* end while */

          /* Send an empty message to indicate EOF (or bad filename) */
          write(msgsock, outbuf, 0);

       } /* end if */

    } while (rval != 0);

    close(msgsock);
    exit(0);

} /* end serveClient*/
```

Biographical Note

David Hansen was born on July 29, 1961 in Sacramento, California where he attended Theodore Judah Elementary School and Kit Carson Junior High School. After moving to Richland, Washington in 1974, he attended Hanford Junior and Senior High Schools, graduating from Hanford High School in 1979. From 1979–1983, David attended Oral Roberts University in Tulsa, Oklahoma, where he began his studies as a music major but graduated in 1984 with a Bachelor of Science degree in Computer Science. That year, he began his career as a computer scientist with the Battelle Memorial Institute at the Department of Energy's Pacific Northwest Laboratory in Richland, Washington. While working at Battelle, David continued his graduate education, attending Washington State University from 1984–1988, graduating in 1988 with a Master of Science degree in Computer Science. In 1990 David began an educational leave of absence from Battelle and enrolled at the Oregon Graduate Institute of Science & Technology (OGI) to pursue his Ph.D. While at OGI, David received a research fellowship from the Oregon Advanced Computing Institute (OACIS).

Publications include:

David M. Hansen. An automatic source-code generator generating subroutines for accessing an Rdb database. In Joel M. Snyder, editor, *Proceedings of the Digital Equipment Computer Users Society*, pages 79–85, Anaheim, CA, December 1987. Digital Equipment Computer Users Society, DECUS.

David Hansen, David Maier, James Stanley, and Jonathan Walpole. An object-oriented heterogeneous database for materials science. *Scientific Programming*, 1(2):115–131, Winter 1992.

Judith Bayard Cushing, David Hansen, David Maier, and Calton Pu. Connecting scientific programs and data using object databases. *Bulletin of the Technical Committee on Data Engineering*, 16(1):9–13, March 1993.

David Maier, Judith Bayard Cushing, David M. Hansen, George D. Purvis III, Raymond A. Bair, D. Michael DeVaney, David F. Feller, and Mark A. Thompson. Object data models for shared molecular structures. In R. Lysakowski, editor, *First International Symposium on Computerized Chemical Data Standards: Databases, Data Interchange, and Information Systems*, Atlanta, GA, May 1993. ASTM.

David M. Hansen and David Maier. Using an object-oriented database to encapsulate heterogeneous scientific data sources. In Jay F. Nunamaker and Ralph H. Sprague, editors, *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume III, pages 408–417, Maui, Hawaii, January 1994. The University of Hawaii, IEEE Computer Society Press.

David Maier and David M. Hansen. Bambi meets Godzilla: Object databases for scientific computing. In James C. French and Hans Hinterberger, editors, Seventh International Working Conference on Scientific and Statistical Database Management, pages 176–184, Charlottesville, VA, September 1994. IEEE Computer Society Press.

R. Venkatesh, D. Hansen, D. Maier and J. T. Stanley II. Applications of object database technology in thermodynamics. In P. Nash and B. Sundman, editors, *Proceedings of Applications of Thermodynamics in the Synthesis and Processing of Materials*, Rosemont, IL, October 1994. The Minerals Metals and Materials Society, TMS.