Computational Chemistry Database Prototype: Objectstore

Judith Bayard Cushing, David Maier, Meenakshi Rao

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-002

June 15, 1991
January 23, 1992 (revised)

Computational Chemistry Database Prototype: ObjectStore

Judith Bayard Cushing David Maier Meenakshi Rao

June 15, 1991
January 23, 1992 (revised)

Abstract

The Computational Chemistry Database Project (CCDB) is a joint effort of computer scientists at The Oregon Graduate Institute and computational chemists and computer scientists at Battelle Pacific Northwest Laboratory's (PNL) Molecular Sciences Research Center and Applied Physics Center. This report describes the database and browser prototypes implemented on a Sun4 Sparcstation 2, using Object Design's Release 1.1 of ObjectStore, and includes an evaluation of ObjectStore as a potential vehicle for object-oriented scientific application systems.

1 Introduction

This report recounts our experience implementing a prototype computational chemistry database using Object Design's Release 1.1 of ObjectStore on a Sun4 Sparcstation 2. The purpose of this work was to explore basic features of ObjectStore and to determine if the system is a candidate for development of the full computational chemistry database. The prototype exercises what we believe is the minimum functionality required for our final system. For more information about the full information model, see "Database Support for Computational Chemistry" [Cus91]. The prototype database was populated with twenty experiments, representing three computational investigations: ethylene, methane and water. Concomitant to this effort, and in order to compare and contrast products, the same design was implemented in three other database systems: Encore, Postgres and GemStone. Three teams of two or three graduate students each completed the other implementations.

In this report, we first briefly outline the Computational Chemistry Database Project (CCDB) and the subset of the information model implemented in the prototype. We then go on to describe ObjectStore and how we used ObjectStore in our project. We consider enhancements to the prototype implementation and describe several aspects of the prototype which we would have done differently if we had known then what we think we know now. Finally, based on this development experience, we critique ObjectStore and evaluate the product as a whole with respect to Zdonik and Maier's object-oriented threshold and reference models [ZM90]. Please bear in mind that this critique of ObjectStore—after only a few weeks of intensive use—is still somewhat premature. A feature whose absence we lament here may be lurking at the turn of the next page in the Reference Manual.

The ObjectStore schema as well as sample data and queries are included in the appendices. The C++ program used to load the database, data used to populate the database, and query programs are available from the authors upon request.

2 The Computational Chemistry Database Project

The Computational Chemistry Database Project is a joint effort of computer scientists at the Oregon Graduate Institute (OGI) and computational chemists and computer scientists at Battelle Pacific Northwest Laboratory's (PNL) Molecular Sciences Research Center and Applied Physics Center. Together with PNL computer scientists D. Michael DeVaney and James Thomas, we have identified *ab initio* computational chemistry (i.e., chemistry from first principles) as an area of initial research from which to explore the applicability of emerging database technology to high performance scientific applications. Our primary domain scientist collaborator at PNL is Dr. David Feller, an active researcher in computational chemistry and himself an author of the computational chemistry program MELDF [DF86, FBD87, FD90, Fel91].

Computational chemistry applications are both computation and data intensive, and have in common with other computational science applications the need both for increasing the speed of calculations and for storing and viewing large amounts of specialized information. In addition, the "laboratory" in which a computational chemist works typically comprises different computationally intensive programs as well as molecular visualization tools, each requiring differently formatted input and producing differently formatted output. Most computational chemistry applications typically run on a number of different architectures and operating systems, with the chemist selecting a target machine for a given experiment based on an estimate of the resources needed. Such a heterogeneous computing environment is common to other computational sciences. Our review of current research indicates that computational chemistry is a good choice for exploring data management problems facing scientific researchers in general [Bel83, BP87, Bel88, BW90, Bur89, Che90, Boa90, Doz90, FB90, GPKF90, HS86, LWS87, LPS90, Olk86b, Olk86a, PL88, SOW84, Gar89, Wat89].

3 Computational Chemistry Information Model

In this section we briefly describe each entity in the subset of the "Computational Chemistry Information Model" which we implemented in the prototype. (See Figure 1.) All entities with the exceptions of laboratory apparatus and molecular template were implemented as persistent C++ classes. However, the prototype database contains far fewer attributes than the full model, in particular for computational results (especially molecular orbitals), molecule, basis set, level of theory and code packages.

Described below are the computational chemistry entities each with its corresponding attributes, as implemented in the prototype database. For the ObjectStore C++ class definitions corresponding to these entities, see Appendix B.

1. **Chemist.** A chemist performs one or more experiments and may be the author of one or more basis sets.
   (a) Name. The chemist's name, a structured text field, e.g., Last, First, MI.
   (b) Address. The chemist's address, usually the laboratory where chemist works.
   (c) Email Address. The chemist's electronic mail address.
2. **Experiment.** An experiment is either a laboratory experiment or a computational chemistry experiment, and may be a collaborative effort of more than one chemist. An experiment produces one or more observable properties for a molecule. This relationship between experiment and a set of observable properties can be modeled by a function which, given an experiment and a property, returns a set of value-unit pairs. (See "observable property", item 10.)

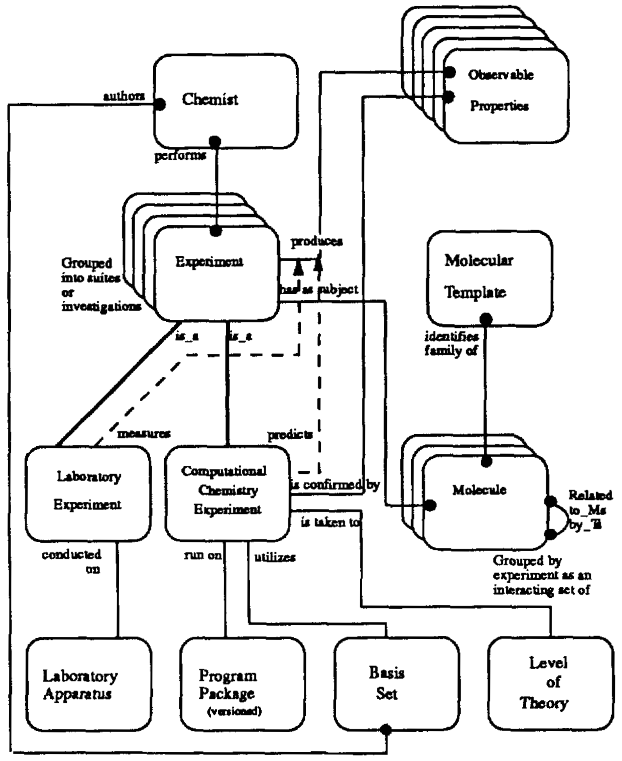

Figure 1: Computational Chemistry Database: Information Model

(a) Name. A textual annotation, or run title, by which a chemist identifies the experiment.

(b) Citation. An unformatted text field (may be null) describing the source of data for this experiment.

(c) Date begun. Initially, only the date that the experiment was actually begun is included in the database. Eventually, this should be a full time stamp so that experiments begun on the same day can be ordered.

(d) Date completed. Again, date completed will eventually be a full time stamp. This field may be null if the experiment is ongoing.

(e) Site. Site where the experiment was performed. This may be different from the performing chemist's address.

3. **Laboratory Experiment.** A laboratory experiment measures one or more observable properties and is conducted on a laboratory apparatus.

4. **Laboratory Apparatus.** A laboratory apparatus is an instrument on which a laboratory experiment is conducted. Eventually, this entity should include additional information about the apparatus that will contribute to the proper interpretation of the experiment, e.g., callibration.

   (a) Instrument. Instrument used to conduct the experiment, e.g., "mass spectrometer".

5. **Computational Chemistry Experiment.** A computational chemistry experiment is run on some code package, using some basis set, and is taken to an appropriate level of theory.

With a large enough basis set and high enough level of theory, molecular properties could be deduced exactly; of course, such a calculation might take months or years, or even be intractible given current computing machinery.

A computational chemistry experiment deduces one or more observable properties, and is confirmed by one or more observable properties (measured by some laboratory experiment). This relationship between a computational chemistry experiment and a set of observable properties can be modeled by a function which, given a computational experiment and a property, returns a laboratory experiment.

(a) Computational Environment Description.
  - i. Computer. The platform on which the experiment run, e.g., Cray2, Sun4.
  - ii. Operating System. The operating system (and version thereof) under which the experiment was run.
  - iii. Code version. The version of the Code Package used to run this experiment. This version number may differ from the current code package version. See item 7b below.
  - iv. CPU time. Processing time, in milliseconds, used by this experiment.
  - v. Elapsed time. Wall clock time, in minutes, elapsed during the time the experiment was running.

(b) Results. The results of a computational chemistry experiment are, in effect, the location of the electrons surrounding the molecule in question, at the molecule's "most stable" configuration, i.e., at the lowest energy.
  - i. Escf. "Self-consistent field energy", a floating point number.
  - ii. Esdci. "Singles and doubles, configuration interaction energy", a floating point number.

6. **Molecule.** A molecule is the subject of one or more experiments.
   - (a) Name. Molecule name, a text field, such as "water", or "ethylene".
   - (b) Chemical Formula. Molecular formula, a text field, e.g., H_20, or C_2H_4.
   - (c) Structure. The location of the atoms in the molecule given as Cartesian coordinates (x, y, z) in angstroms, the atomic mass and charge, for each atom in the molecule.

7. **Code Package.** A code package is a computational chemistry application program or programs on which a computational chemistry experiment is run.
   - (a) Name. Name of the code package, e.g., "Gaussian", "GAMESS".
   - (b) Version. The database will always return, unless otherwise specified, the object representing the version of the code package in current use. Code package versions on which particular experiments were run are also stored in the database. (See item 5(a)iii.) There will be one code package instance per software publisher's version, per platform type, per compiler used to compile that version. Additional meta information required for interpretation of experimental results will be stored in the full project database.
     - i. Code Version Name. Name given to the version by the publisher.
     - ii. Available On. Platform on which this version runs.
     - iii. Compiler Version. Language, version and release of the compiler used to compile the code.

iv. Available Date. Date this version of the code became available. May be null if not yet available.

v. Archived Date. Date this version of the code was archived. May be null.

8. **Basis Set.** The selection of the basis set is critical not only to how efficiently a computational experiment will run, but also to its correctness. Eventually, the CCDB database will probably interface with a basis set library, which will contain more detailed information on basis sets. Much more detailed information than is given below is required by most computational chemistry programs and (in any case) for experimental comparability.

   (a) Name. e.g., STO-3G, Dunning DZP, 4-31G, 6-311G, 4-31G*.

9. **Level of Theory.** The levelof theory is an input parameter that specifies the degree of specificity and accuracy to which an experiment should be taken.

   (a) Name. e.g., MP2, MP3.

10. **Observable Property.** An observable property is basically a property-unit-value triple, represented as two text fields (for the name of the property described and the units in which its value is give) and one floating point number (representing the value of the property). The database includes, but is not limited to, the following properties: hydrophobicity, polarizability, hyper-polarizability, and anisotropicity.

4 CCDB Sample Queries and Delete Operations

In this section we briefly specify the queries and operations which we chose to implement in the prototype. The queries described below should each retrieve a set of experiments, to be displayed using either a terse or verbose format, as specified. A terse display should present only identifying data for the set of experiments, preferably one experiment per line in a tabular form, e.g., performing chemist(s), molecule (name and formula), code (name), basis set (name), and level of theory (name). A verbose display should present all data about that experiment including results.

The six prototype queries are:

1. **Display experiments.** This query prepares a terse-display of experiment(s) performed by a given chemist, identified by name.

2. **Display experiments ordered by date.** This query prepares a terse-display of all experiments, but also includes the date each experiment was initiated. The display is ordered by date in most-recent-first order.

3. **Display experiments for a given molecule.** This query prepares a terse-display of the experiments for a given molecule. An optional extension would allow the user to limit the search to specified observable properties, or to laboratory or computational experiments only.

4. **Create a collection of experiments.** This query is in effect a manual selection facility allowing the user to browse a set of experiments, from which a new collection of experiments is created. After displaying in verbose form each experiment in a given set, it allows the user to indicate whether that experiment should be included in the new collection. Once each member of the old set is displayed, the user can name the new collection.

5. **Delete an experiment from the database.** When deleting an experiment from the database, all fields in other objects that refer to it must be set to null. "Orphaned objects" should also be deleted, for example, in the case where a particular observable property is referred to by just one computational experiment, and that experiment is deleted. If the database system does not maintain referential integrity and provide garbage collection for unreachable objects, the application should do so.

6. **Delete a basis set from the database.** Referential integrity should be preserved, as above.

5 Object Design's ObjectStore

ObjectStore [Obj] implements a client-server database model, providing persistence by storage class either through its DML/DDL extensions to C++ or through a C++ library interface. An ObjectStore database can be distributed onto several workstations. Three processes must be running to support an application: "ossvr" (one ObjectStore database server for each workstation client); "osdirman" (at least one directory manager per site to maintain the Objectstore namespace); in addition, one ObjectStore cache manager is required for each workstation on which client applications run.

Objects are declared persistent through the use of ObjectStore's parameterization of the C++ "new" command. Navigation in the database to persistent objects is performed by starting at a persistent variable (known in ObjectStore as a "root" variable). Because objects are declared persistent via allocation, persistence is orthogonal to type. This important characteristic of ObjectStore means that the in-memory structure of persistent objects can be the same as that for nonpersistent objects and, hence, allows the use of existing C or C++ code with ObjectStore. The developer can choose to use ObjectStore's extensions to C++ (the Data Manipulation Language or DML), implemented via a C++ preprocessor (itself a preprocessor to C) or a library interface (with which developers can use their preferred C++ compiler).

In addition to the C++ preprocessor and libraries, ObjectStore supplies database utilities and development tools, including a graphical schema designer, a database browser (that can be used to navigate from persistent variables to persistent objects and includes a basic interactive query facility), and an interface to the GNU debugger.

6 Project Development

The initial prototype development effort spanned about four weeks. Two programmers, working about three-quarters time, started from an ObjectStore independent C++ program designed to load the data into C++ in-memory structures. At the onset of the project, neither had significant experience programming in C++, and one had a cursory knowledge of ObjectStore. We successfully populated the database with all 20 experiments and implemented the specified queries.

6.1 Developing the Schema

We used ObjectStore's Schema Designer to generate a simple schema with classes and relationships between classes. To add attributes and methods, and for all schema modifications, we simply edited the C++ ".hh" file.

6.2 Populating the Database

We opted to automatically load the database using C++ iostream libraries, converting to an ObjectStore program a C++ program we wrote to read a highly modified version of the input data. This approach helped us isolate ObjectStore issues from those of C++ and nicely illustrated the functionality ObjectStore provides over C++. Automatically loading the database from data files required tedious by-hand modifications to the input data as well as a high front-end programming effort. However, once working, the automatic load made it easy to modify the schema, refine the input datasets, and reload the database.

An enhancement to the system on which we are currently working is an interactive object entry and modification routine to generate test data and to fix minor data errors without explicitly deleting and reloading an entire computational experiment or set of experiments. As the prototype now stands, we must delete and reload by hand every object related to a given chemist's computational experiments simply to change the email address of that chemist. Providing a better object entry and modification facility for a given application involves writing specific methods to change object properties and is programming intensive. A welcome addition to ObjectStore would be a rudimentary object editor as an extension to the browser.

The somewhat troublesome issue of value identity vs. object identity arose in checking for and excluding duplicate objects. When loading the database, we decided not to create a new chemist, lab experiment, property, basis set, code package, level of theory or observable property if an object with "equal values" already existed in the database. Atom objects were duplicated if included in different molecules. In order to determine if a referred to object was "equal in value" to an object already existing in the database, we used a primary-key-like attribute (serial number). This artificially generated attribute is not a viable long term solution.

ObjectStore's set collection facility makes it easy to check for (shallow) object identity, but to check for value identity, each attribute in the new object must be checked against each attribute in the existing persistent object. However, even this is probably too stringent a requirement. A chemist who moves from PNL to LNL is still the same chemist. Even more to the point: "Dave Feller" is (probably) the same chemist as "David Feller". Something akin to the upfront relational discipline of defining unique primary keys as tuple identifiers cannot be avoided in object-oriented systems, and carries some design and implementation cost.

6.3 Using the Browser to Verify the Database

We used the ObjectStore browser extensively in our development work, and found it extremely helpful. Indeed, the browser worked so well, we never felt compelled to learn the GNU debugger.

6.4 Query Implementation

Implementing the prototype queries using the DML interface in ObjectStore was straightforward. ObjectStore enforces referential integrity constraints as long as named inverse relationships are defined in the schema. The six database operations described in Section 4 are categorized below into those which (1) display experiments, (2) create a new collection, and (3) delete experiments:

1. Displaying experiments by chemist, date and molecule. These queries were implemented using ObjectStore's set iterator "foreach". ObjectStore provides support for ordering a display by date through a general sorting capability. Ranking functions are supplied for basic types, and users can write rank functions for user-defined types. The order of the set iterator "foreach"

can be controlled using the "index path" option, and either ascending or descending order can be specified.

An ObjectStore alternative to using the "foreach" set iterators is to specify a query declaratively using the "bracket-notation". (See item 8 in Section 8.4.)

2. Creating a collection of experiments. Viable options in ObjectStore for creating a new collection of objects with a dynamically specified, user-defined name include:

   (a) Create a persistent class to act as a kind of umbrella class. This class includes as instances the set of user-supplied names and pointers to the new, user-defined, persistent collections. For example, define an umbrella class named "personal experiments"; each instance of "personal experiments" is a set of experiments. Suppose a chemist is interested in comparing computational experiments performed on ethane that use basis sets authored by Dunning. The query would create a new instance of "personal experiments", with a user supplied name, e.g., "ethane experiments using Dunning basis sets". Into this set, the query would place pointers to the persistent experiments of interest, as specified by the user.

   (b) Use the user-supplied name as a persistent variable containing a pointer to the new collection.

   (c) Create a new and separate database for the specified set of experiments, copying into the new database all the referenced objects.

   (d) Same as option 2c above, but use the ObjectStore cross-database reference mechanism so that objects referred to need not be copied.

   Option 2a was chosen to implement the creation of a new collection. We preferred option 2a to option 2b because option 2a isolates user-generated persistent variables to the "user's experiment set" class, while option 2b clutters the database with user-defined names and complicates the maintenance of referential integrity. We chose option 2a over option 2c because it is more space efficient and does not require duplicating objects, even though it does not provide the capability of creating a separate database that could be shipped to a different site, as option 2c would. We did not take time to pursue option 2d, cross-database references.

   An enhancement to our implementation of this query would be to allow the user to add, modify, or delete experiments from his or her personal set of experiments without affecting experiments in the public database.

3. Deleting experiments and basis sets from the database.

   Because the Computational Chemistry Information Model did not specify a named relationship from basis set back to chemist (and we did not implement this inverse relationship), ObjectStore does not provide integrity checking when a basis set is deleted. Deleting a basis set thus causes "invalid persistent pointers" in the chemist and computational experiment objects.

   Short of including the inverse relationship pointers, or programmatically disallowing the deletion of a basis set referred to by an experiment, one could maintain referential intergrity by creating a new attribute for basis set: inactive status. A "delete basis set" query on a basis set that refers to a computational experiment would render that basis set inactive. A "list basis set" query would not list inactive basis sets, but a query of computational experiments will indicate that the associated basis set is inactive. We much prefer the overhead of maintaining inverse relationships to this alternative.

7 Lessons Learned and Future System Enhancements

Because we were relatively inexperienced C++ programmers and time was short, we did not take full advantage of encapsulation, constructors, display methods, and virtual functions. For example, for ease of implementation, we chose to use only public classes. Since "private" or "protected" classes were not used, we had no need for using the "friend" mechanism. Future enhancements include encapsulation of class definitions, moving much of the constructor and display functionality from the population program and query programs into the schema, and providing access methods for each class. Also because of the programmers' inexperience with C++, we did not implement application class libraries.

There is room for significant work in crafting a user interface through which the user would invoke queries with a window-like system, pop-up menus, etc. Graphical representation of basis sets, energy levels, and molecular structure would be particularly helpful. See the discussion of tools in Section 8.2.

8 Preliminary Evaluation of ObjectStore

All in all, we had a very positive experience using this product, and were able to complete a database prototype relatively painlessly in a short period of time. Minimal set up is required to get up and running, and the basic tools supplied are reliable and easy to learn. The DML interface is in our opinion a straightforward and sensible extension to C++, and we highly recommend it over the library interface.

8.1 Installing ObjectStore

Object Design supplies a fairly extensive set of tutorial programs that can be used, not only to learn ObjectStore, but also as a test of the product installation; we found these quite helpful in testing the installation. The Installation Guide/Release Notes and Tutorial give simple instructions to users for getting started quickly, which is straightforward for the UNIX C++ programmer familiar with makefiles or for a programmer with some understanding of UNIX and C and a good understanding of another database system. Using ObjectStore, even the query generator, is not (yet) for the nonprogrammer.

8.2 Development Tools

Our experience with ObjectStore development tools was quite positive. They were easy to learn (with the possible exception of the debugger), reliable and robust. Some additional functionality would be helpful; for example, the schema designer and browser default automatically to the library interface option. Because of significant differences in syntax and presentation between the DML and library interfaces, it would be very useful if the user could permanently specify the default to DML. The inability to set a default option ranges from a minor inconvenience in the case of the browser to a major source of confusion for inexperienced users in the case of the schema designer.

1. Database User Utilities. ObjectStore is released with a UNIX-like set of utilities for database administration: e.g., "osls" to list ObjectStore databases. Those most important to the beginning user are quite intuitive for any UNIX user.

2. SchemaDesigner. This tool is easy to learn for someone who has used either a "Macintosh-like" windows application or Smalltalk. The graphical interface works nicely for first cut definition of classes and relationships. However, going through the schema designer to define attributes and classes seemed somewhat cumbersome and we chose instead to use an ordinary text editor. The schema designer holds the user's schema in an ".sd" file, and generates C++ ".hh" file upon request. Unfortunately, however, the schema designer reads only its ".sd" file. Thus, once the programmer has modified the ".hh" file, ObjectStore's schema designer tool is no longer of much use. ObjectStore is working on this problem, as well as on making schema information available to programs.

   An easy way of printing a graphical schema would be nice.

3. The Browser. This tool was also easy to use and a real help in debugging our database population program. The user can very easily navigate through the database by clicking on pointer values in a displayed object. Database values are displayed in easy to read formats. The browser displays both class structures and data values. One can also browse the ObjectStore generated compilation schema.

   A rudimentary interactive query processor is available within the browser. A user can easily specify boolean queries, e.g., retrieving chemist objects with (address == PNL). Booleans can be combined, but no methods can be accessed through the query processor.

   Leaving the browser up during compilation and execution of programs was somewhat problematical — even when the compile did not modify the schema. Closing the database from the browser during compilation was not sufficient to prevent the browser from getting confused, especially after schema modification.

   Options "Print Visible Panes" and "Print [a particular pane]" produce Xwindows data. An easier way of printing and dumping objects would be nice.

4. The Debugger. Object Design inlcudes with ObjectStore version 3.2 of the GNU Source-Level Debugger (GDB), extended to interface with the DML version of ObjectStore. Unfortunately, we were familiar only with DBX and there just wasn't time to learn GDB. The ObjectStore user (as users of many other object-oriented database systems) must choose between using the DML and his or her favorite C++ compiler and toolchest.

An applications generator like "ET++" [WGM89] would be particularly helpful in implementing a full blown database browser for users. Many of the CCDB objects such as basis sets and molecular structures should be represented graphically.

8.3 Threshold Model

We used Zdonik's and Maier's criteria for object-oriented databases [ZM90] to evaluate ObjectStore's capabilities. Our prototype application effort indicates that ObjectStore definitely meets the requirements of the object-oriented threshold model:

1. Database Functionality.

ObjectStore provides essential database features including model and language (C++ structures and added collections) and binary unattributed relationships. Persistence is provided via allocation and concurrency is supportted through user specified transactions, including rollback on transaction abort (which we witnessed in populating our database). The size of an ObjectStore database is not limited by the amount of main memory nor the address range of virtual memory.

2. Object Identity. Every database object is assigned a unique object identity. Indeed, object identity can be preserved across distributed databases.

3. Encapsulation. ObjectStore supports and enforces encapsulation through standard C++ features such as public, private, and friend.

4. Complex State. An ObjectStore store object can contain references to other ObjectStore objects, which in turn contain references to still other ObjectStore objects. ObjectStore provides facilities for user-defined types, subtyping and inheritance, and unattributed binary relationships.

8.4 Reference Model

ObjectStore exhibits seven of the ten characteristics of Zdonik's and Maier's reference model.

1. Structured representations for objects. C++ schemas constitute structured object representation.

2. Persistence by Reachability. Persistence in ObjectStore is by storage class and is orthogonal to type. Persistent objects are reachable through persistent variables, but may not point to transient objects outside transaction boundaries. ObjectStore apparently does not garbage collect, thus allowing unreachable objects to remain in the database heap.

3. Typing of objects and variables is done at both compile and run time. The C++ typecast operator is used for subtyping at run time. For example, for an experiment to "know" its type, one must supply a virtual function in the supertype with a corresponding function in the subtypes and use C++ typecasting.

4. Three hierarchies. The C++ implementation hierarchy supported by ObjectStore subsumes a specification hierarchy on types and a hierarchy of representations and methods. It appears to us that a classification hierarchy on explicit collections of objects would have to be implemented explicitly by a user; there is no triggering mechanism to support such an implementation.

5. Polymorphism. C++ (and hence ObjectStore) supports polymorphism through dispatching. For example, a given class and some of its subclasses may each have display functions "display". Display is thus part of the protocol of every subclass of the class. Which display function is invoked on a given call to display depends on the type of the actual object to be displayed. In addition, ObjectStore supports parameterized types, virtual functions for persistent as well as transient objects, and function overloading.

6. Collections. ObjectStore supports sets, bags and lists. Furthermore, a user can specify for any given collection a physical representation policy, e.g., B-trees or arrays.

7. Name spaces. Persistent objects are accessible either directly through persistent variables or by navigation from another persistent object accessed through a persistent variable. Persistent variables, known in ObjectStore as "root variables" are declared as such and associated with a database by using the keyword "persistent",

8. Queries and indices. ObjectStore supplies a query facility through "foreach" iterators, an extension of C++. A query represented using a bracket notation is an alternative optimizable by creating an index. For example, where os_Set<person*> &people, the set of teenagers in a person database can be determined in two alternative ways, the latter of which is an optimizable shorthand method of the former.

   -
     ```
     os_Set<person*> &teenager;
        foreach(person *p, people)
            if (p->age >= 13 && p->age <= 19) teenager |= p;
     ```

   -
     ```
     os_Set<person*> &teenagers =
     people[: this->age 13 && this->age <= 19 :];
     ```

   ObjectStore's clustering facility provides an additional opportunity for enhancing the movement of data from disk to memory. New persistent objects can be placed explicitly in the same segment as existing persistent objects. The user can programmatically specify whether data is transerred to the cleint cache a page at a time or a segment at a time.

9. Relationships. Named relationships (including inverse relationships and some integrity checking) are nicely supported via extensions to the C++ or a library interface. As of this point in time, only unattributed binary relationships are supported.

10. Versions. Through its configuration and workspace classes, ObjectStore provides for accessing versions of an object's state and for assembling configurations of consistent versions of objects. We did not explore this capability.

8.5 Final Observations about ObjectStore

Those familiar with relational systems might well remark that this system, probably like all object-oriented systems, is as yet somewhat immature:

1. The DML interface is, in our opinion, much cleaner and easier to use than the library interface. However, examples of advanced features included in the new 1.1 User's Guide seem slightly biased towards the library interface, as do some tools such as the Browser and Schema Designer that default only to the library interface option. Will ObjectStore continue to support both a DML and a Library interface?

2. The ObjectStore compiler (actually a C++ preprocessor), necessary for the DML interface, delivers some unhelpful error messages, e.g., "syntax error".

3. Loading our relatively small database seemed slow and there was a noticeable startup for queries. In fairness we note that for ease of implementation we declared the extent of every object type as a persistent root variable and did no performance optimization whatsoever. Once objects were in memory, i.e., after warmstart, access seemed extremely fast.

4. There is no support for generating a user interface, nor is an extensive class library supplied. Only the (rudimentary) C++ string classes are available.

9 Acknowledgements

This work is supported financially by grants from the Oregon Advanced Computing Institute (OACIS), Battelle Pacific Northwest Laboratory's Molecular Sciences Research Center and Applied Physics Center, and NSF Grant IRI-9117008. Grants from Object Design, Inc., and Servio Corporation generously supplied the database systems for the ObjectStore and the GemStone prototypes respectively.

The authors would also like to thank the developers of Postgres and Encore for making their software available in the public domain. Robert Kelley of Sequent Computers contributed the original C++ program used as a basis for the ObjectStore database loading program.

A Sample Data Used to Populate the Database

```
ComputationalExperiment #1
performedBy
+
Chemist #1
name Dave Feller
address PNL
email   feller@pnl
$ //ends Chemist #1
$ //ends performedBy
name Ethylene DZP Test Case with no compression of integrals
citation PNL Basis Set Library
dateBegun 12/12/90
dateCompleted 12/12/90
site PNL
cpu  45.44 sec
cpuElapsed 49.00 sec
escf -78.0505295623
esdci   -78.3281048
hasAsSubject
Molecule #1
name ethylene
formula C2H4
structure
+
atom C
mass 12.01115
charge  6.0
x 1.25666814
y 0.0
z 0.0
$
+
atom C
mass 12.01115
charge  6.0
x -1.25666814
y 0.0
z 0.0
$
+
atom H
mass 1.00797
charge 1.0
x 2.32513368
y 1.72999314
z 0.0
$
+
atom H
mass 1.00797
charge 1.0
x -2.32513368
y 1.72999314
z 0.0
$
```

```
+
atom H
mass 1.00797
charge 1.0
x 2.32513368
y -1.72999314
z 0.0
$
+
atom H
mass 1.00797
charge 1.0
x -2.32513368
y -1.72999314
z 0.0
$ //ends Atom
$ //ends structure
$ //ends Molecule #1
codePackage
CodePackage #1
name MELDF
codeVersion 2.0
computer Sun4
language Fortran 1.2
dateAvailable 1/1/90
dateArchived  0/0/00
$ //ends CodePackage #1
basisSet
BasisSet #1
name Dunning DZP
authoredBy
+
Chemist #2
name Jim Dunning
address PNL
email   dunning@pnl
$ //ends Chemist #2
$ //ends authoredBy
$ //ends basisSet
levelOfTheory
LevelOfTheory #1
name MP2
$ //ends levelOfTheory #1
produces
+
Property #1
property polarizability
unit A**3
value 2.2
$ //ends Property #1
$ //ends Property #1
isConfirmedBy
+
LabExperiment #1
performedBy
+
Chemist #3
name Belkis Izer
```

```
address Cornell
email izer@cornell
$ //ends Chemist #3
$ //ends performedBy
name ethylene polarizability
citation Campbell, Chemical Systems
dateBegun 0/0/00
dateCompleted 2/5/69
site Istanbul, Turkey
instrument parallel plate condensor
produces
+
Property #2
property polarizability
unit A**3
value 2.1
$ //ends Property #2
$ //ends Property #2
$ //ends produces
$ //ends labExperiment #1
$       //ends isConfirmedBy
$ //ends produces
$ //ends ComputationalExperiment #1
```

B Class Definitions: ObjectStore Schema

```
/*  Computational Chemistry ObjectStore Prototype
    May, 1991

 * Schema header file for ossd file 'prj'.
 * Produced by ossd on Tue May  7 17:42:40 1991
 *
 * Classes and their forward declarations are written
 * in base class to derived class order, as is necessary
 * for compilation.
 */

#include <sys/types.h>
#include <string.h>

struct Date {
    short mm, dd, yyyy;
    Date() : mm(0), dd(0), yyyy(0) {}
    friend istream& operator>>(istream& s, Date &d);
};

istream& operator>>(istream& s, Date &d)
{
char c;
return s >> d.mm >> c >> d.dd >> c >> d.yyyy;
}

#define LAB  1
#define COMP 2

typedef char*  String;

extern database *db;

class Chemist;
class Experiment;
class LabExperiment;
class CompExperiment;
class Property;
class BasisSet;
class LevelofTheory;
class CodePackage;
class Molecule;
class Atom;
class PersonalCE;

static char buffer[1024];


class Chemist {
  /* A chemist performs experiments. */
  public:
    persistent<db> os_Set<Chemist*> extent;
    os_Set<Experiment*> performs inverse_member isPerformedBy;
    os_Set<BasisSet*> authors inverse_member isAuthoredBy;
    int    id;
    char* name;
    char* address;
    char* email;
    Chemist (int Chemist_id) {
id = Chemist_id;
        extent.insert(this);
}
    ~Chemist () {
        extent.remove(this);
}
};


class Experiment {
  /* An experiment is either computational or laboratory. */
  public:
    os_Set<Chemist*> isPerformedBy inverse_member performs;
    os_Set<Property*> produces inverse_member isProducedBy;
    Molecule* hasAsSubject inverse_member isSubjectOf;
    int id   indexable;
    char* name;
    char* citation;
    Date begun indexable;
    Date  completed;
    char* site;
    virtual int WhatAmI() = 0;
    Experiment (int Experiment_id) {
id = Experiment_id;
}
};


class LabExperiment : public Experiment {
  /* A LabExperiment confirms a computational experiment. */
  public:
    persistent<db> os_Set<LabExperiment*> extent;
    char* instrument;
    os_Set<CompExperiment*> confirms inverse_member isConfirmedBy;
    virtual int WhatAmI() {
return LAB;
}
    LabExperiment (int id) : Experiment(id) {
extent.insert(this);
}
    ~LabExperiment(){extent.remove(this);}
};


class CompExperiment : public Experiment {
  /* A Computational Experiment uses a Code Package and Basis Set to some Level of Theory. */
  public:
    persistent<db> os_Set<CompExperiment*> extent;
    BasisSet* usesBS ;
    LevelofTheory* isTakenTo ;
    CodePackage* usesCP ;
    LabExperiment* isConfirmedBy inverse_member confirms;
    int id;
    float cpuTime;
    float  elapsedTime;
    float escf;
    float esdci;
    virtual int WhatAmI() {
return COMP;
}
    CompExperiment (int s)
: Experiment(s) {
        id=s;
extent.insert(this);
}
 ~CompExperiment () {
        extent.remove(this);
          }
};


class Property {
  /* A property is a property/unit/value triple. */
  public:
    persistent<db> os_Set<Property*> extent;
    Experiment*  isProducedBy inverse_member produces;
    int id;
    char* name;
    char* unit;
    float value;
    Property (int s) {
        id=s;
extent.insert(this);
}
    ~Property () {
extent.remove(this);
}
};


class BasisSet {
  /* A basis set is used in a computational experiment. */
  public:
    persistent<db> os_Set<BasisSet*> extent;
    os_Set<Chemist*> isAuthoredBy inverse_member authors;
    int id;
    char* name;
    BasisSet (int i) {
name = 0;
id=i;
extent.insert(this);
}
    ~BasisSet ()  {
delete name;
extent.remove(this);
}
};


class PersonalCE {
 /* Users can define their own sets of Comp Exp */
   public:
   persistent<db> os_Set<PersonalCE*> extent;
```

```
    char*       name;
    os_Set<CompExperiment*> my_set;
    PersonalCE (char* s){
    name= new(db) char[strlen(s)+1];
    strcpy(name,s);
    extent.insert(this);
    }
   ~PersonalCE () {extent.remove(this);}
    };


class LevelofTheory {
  /* A computational chemistry experiment is taken to a level of theory. */
  public:
    persistent<db> os_Set<LevelofTheory*> extent;
    int id;
    char* name;
    LevelofTheory (int i){
id=i;
extent.insert(this);
}
    ~LevelofTheory () {extent.remove(this);}
};


class CodePackage {
  /* A code package is used by a computational experiment. */
  public:
    persistent<db> os_Set<CodePackage*> extent;
    int id;
    char* name;
    char* codeVersion;
    char* computer;
    char*       compilerVersion;
    Date dateAvailable;
    Date dateArchived;
    CodePackage (int n) {
id = n;
extent.insert(this);
}
    ~CodePackage() {extent.remove(this);}
};


class Molecule {
  /* A molecule is the subject of an experiment. */
  public:
    persistent<db> os_Set<Molecule*> extent;
    os_Set<Atom*> hasAtoms ;
    os_Set<Experiment*> isSubjectOf inverse_member hasAsSubject;
    int   id;
    char* name;
    char* formula;
    Molecule (int i){
id=i;
extent.insert(this);
}
    ~Molecule () {extent.remove(this);}
```

```
};


class Atom {
  /* An atom is a component of a molecule. */
  public:
    char* name;
    float mass;
    float charge;
    float x;
    float y;
    float z;
    Atom (char* str){
name= new(db) char[strlen(str)+1];
strcpy(name,str);
}
    ~Atom () {}
};
```

C Sample Query Program

```
/*  Computational Chemistry ObjectStore Prototype
    May, 1991

    Query1:  Terse display all experiments performed by a given chemist

    dexp /ccdb/db1 "Dave Feller"
        The datbase name & chemist name are passed via parameters.
    */

#include <stream.h>
#include <string.h>
#include <assert.h>

#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/relat.hh>

#include "ccdb.hh"

database *db = 0;
main(int nargs, char **argv)
{
  char  name[30];
  char  basisname[20];

  printf("Terse-display experiments performed by:  %s\n\n", argv[2]);
  do_transaction() {
    os_Set<CompExperiment*> all_CompExp;
    os_Set<Chemist*> all_chem;
    Chemist* chem;
    CompExperiment* compex;
    all_CompExp = CompExperiment::extent[:1:];
    foreach (compex, all_CompExp)
      {foreach (chem, compex->isPerformedBy)
         {if (!strcmp(chem->name, argv[2]))
            {// terse display the experiment
     strcpy(name, compex->name);
     name[30] = '\0';
     printf("%32s%6s%8s%15s%8s\n",
                       name,
               compex->hasAsSubject->formula,
               compex->usesCP->name,
       compex->usesBS->name,
               compex->isTakenTo->name );
   }
}
     }
  db->close();
  }
}
```

D Sample Query Output

```
Terse-display experiments performed by:  Dave Feller

   EXPERIMENT NAME                MOL.  CODE     BASIS SET     LEVEL

  Methane Test Case with no comp   CH4   MELDF    Dunning DZP     MP2
  Ethylene (vary Basis Set)       C2H4   MELDF            DZP     MP2
  Methane (vary level of theory)   CH4   MELDF    Dunning DZP     MP3
  Ethylene DZP Test Case          C2H4   MELDF    Dunning DZP     MP2
        Methane (vary basis set)   CH4   MELDF         4-31G      MP2
  Ethylene (vary Basis Set)       C2H4   MELDF        STD-SET     MP2
  Ethylene (vary Basis Set)       C2H4   MELDF          4-31G     MP2
   Ethylene (optimize structure)  C2H4   MELDF    Dunning DZP     MP2
  Methane Test Case (vary Basis    CH4   MELDF          3-21G     MP2
      Ethylene (vary LoT to MP3)  C2H4   MELDF    Dunning DZP     MP3
   Ethylene (optimize structure)  C2H4   MELDF    Dunning DZP     MP2
  Ethylene (vary LoT) DZP Test C  C2H4   MELDF    Dunning DZP     MP1
```

References

[Bel83] J. L. Bell. *Data Structures for Scientific Simulation Programs.* PhD thesis, University of Colorado, Boulder, CO, 1983.

[Bel88] J. L. Bell. A specialized data management system for parallel execution of particle physics codes. In *Proceedings ACM SIGMOD*, volume 18, pages 277–285. ACM, ACM Press, June 1988.

[Boa90] Computer Science and Technology Board. *Computing and Molecular Biology: Mapping and Interpreting Biological Information, a CSTB Workshop.* NRC, Washington, DC, 1990.

[BP87] J. L. Bell and G. S. Patterson, Jr. Data organization in large numerical computations. *The Journal of Supercomputing*, 1:105–136, 1987.

[Bur89] C. Burks. Genbank: Current status and future directions. Technical Report LA-UR-89-1154, Los Alamos National Laboratory, Los Alamos, NM, April 1989.

[BW90] A. J. Bleasby and J. C. Wootton. Construction of validated, non-redundant composite protein sequence databases. *Protein Engineering*, 3(3):153–159, 1990.

[Che90] R. M. Chervin. High performance computing and the grand challenge of climate modeling. *Computers in Physics*, pages 234–238, May/June 1990.

[Cus91] Judith Bayard Cushing. Database support for computational chemistry. Technical report, The Oregon Graduate Institute, Beaverton, OR, 1991.

[DF86] E. R. Davidson and D. Feller. Basis set selection for molecular calculations. *Chemical Review*, 86:681–696, 1986.

[Doz90] J. Dozier. Looking ahead to EOS: The earth observing system. *Computers in Physics*, pages 248–259, May/June 1990.

[FB90] J. W. Fickett and C. Burks. Development of a database for nucleotide sequences. In M. S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 1–35. CRC Press, 1990.

[FBD87] D. Feller, C. M. Boyle, and E. R. Davidson. One-electron properties of several small molecules using near Hartree-Fock limit basis sets. *Journal of Chemical Physics*, 86(6):3424, 1987.

[FD90] D. Feller and E. R. Davidson. Basis sets for *ab initio* molecular orbital calculations and intermolecular interactions. In K. B. Lipkowitz and D. B. Boyd, editors, *Reviews in Computational Chemistry*, pages 1–43. VCH, New York, 1990.

[Fel91] D. Feller. *MELDFX User's Guide.* Molecular Science Research Center, Battelle Pacific Northwest Laboratories, Richland, WA, April 1991.

[Gar89] J. M. Thornton and S. P. Gardner. Protein motifs and database searching. *TIBS*, 14:300–304, July 1989.

[GPKF90] P. M. D. Gray, N. W. Paton, G. J. L. Kemp, and J. E. Fothergill. An object-oriented database for protein structure analysis. *Protein Engineering*, 3(4):235–243, 1990.

[HS86] W. D. Hillis and G. Steele, Jr. Data parallel algorithms. *CACM*, 29(12):1170–1183, December 1986.

[LPS90] S. Letovsky, R. Pecherer, and A. Shoshani. Scientific data management for human genome applications. In Z. Meral Ozsoyoglu, editor, *Data Engineering (Special Issue on SSDBMS)*, volume 13, page 51, Washington, DC, September 1990. IEEE Computer Society.

[LWS87] R. H. Lathrop, T. A. Webster, and T. F. Smith. ARIADNE: Pattern-directed inference and hierarchical abstraction in protein structure recognition. *CACM*, Vol 30, No 11:909–921, November 1987.

[Obj] Object Design, Burlington, MA. *ObjectStore.*

[Olk86a] F. Olken. Physical database support for scientific and statistical database management. Technical Report LBL-19940 (rev2), Lawrence Berkeley Laboratories, Berkeley, CA, May 1986.

[Olk86b] F. Olken. Scientific and statistical data management research at LBL. Technical Report LBL-21623, Lawrence Berkeley Laboratories, Berkeley, CA, June 1986.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. (U.S.)*, 85:2444–2448, April 1988.

[SOW84] A. Shoshani, F. Olken, and H. K. T. Wong. Characteristics of scientific databases. *Proceedings of the Tenth International Conference on VLDB*, pages 147–159, August 1984.

[Wat89] M. Waterman. Foreword. *Bulletin of Mathematical Biology*, 51(1):1–4, 1989.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of et++, a seamless object-oriented application framework. *Structured Programming*, 2:1–25, 1989.

[ZM90] S. B. Zdonik and D. Maier, editors. *Readings in Object-oriented Database Systems.* Morgan Kauffman, San Mateo, CA, 1990.