Making Smalltalk a Database System

George Copeland
Servio Logic Corporation

David Maier
Oregon Graduate Center
and
Servio Logic Corporation

Abstract
To overcome limitations in the modeling power of existing database systems and provide a better tool for database application programming, Servio Logic Corporation is developing a computer system to support a set-theoretic data model in an object-oriented programming environment. We recount the problems with existing models and database systems. We then show how features of Smalltalk, such such as operational semantics, its type hierarchy, entity identity and the merging of programming and data language, solve many of those problems. Nest we consider what Smalltalk lacks as a database system: secondary storage management, a declarative semantics, concurrency, past states. To address these shortcomings, we needed a formal data model. We introduce the GemStone data model, and show how it helps to define path expressions, a declarative semantics and object history in the OPAL language. We summarize similar approaches, and give a brief overview of the GemStone system implementation.

1. Introduction

The areas of database systems, programming languages and artificial intelligence already have large overlaps in some areas [PP]. Designing a state-of-the-art system in one field means employing ideas from the other two fields. Databases require better integrated application programming interfaces, expert systems must deal with large collections of base facts, and programming languages need richer ways to model their data. Current database systems are primarily an effort to implement an abstract data type over the memory of a machine, rather than to support easy and natural modeling of real-world enterprises. They hide the complexities of file systems and indexing techniques, and provide a degree of physical data independence. The next generation of database systems will be knowledge management systems, with more support for data semantics, inferencing and general purpose programming [DE].

Servio Logic Corporation is designing a programming system for database applications, presently called GemStone, that avoids the restrictions and problems of current commercial database systems. The GemStone language, OPAL, incorporates ideas from knowledge representation, abstract data types, and object-oriented programming, as well as semantic data languages, set-theoretic data models, non-procedural query languages and temporal semantics. It provides rich data modeling facilities with a natural interface to a high-level programming language. This paper considers the choice of Smalltalk-80* [GR] as a basis for the OPAL language, and outlines the enhancements we made to Smalltalk to derive the OPAL language and GemStone system.

2. Shortcomings of Commercial Database Systems

We outline, in this section, the limitations of existing database systems in the areas of data modeling and programming interfaces. We also set forth the GemStone design goals that address each limitation.

A. Type Definition Facilities: Most database systems supply a fixed set of simple types–integer, real and character string–and perhaps a few specialized types, such as date or money. However, they lack the ability to define new simple types and to add operations to the existing types. We can't create a new "employee number" type with a non-standard ordering, or a new operator "nearest payday" on dates, without going outside the database system to an application programming language. The constructors for higher-level types are also limited. The relational model supports "tuple" and "set of homogeneous tuples" (relation). The hierarchical model supports "segment" and "tree of segments". The network model has "record" and "owned list of record" (CODASYL set). A value in a record itself cannot be a structured data item, except for limited support of repeating fields. The operations for higher-level types are induced by the type constructors and cannot be extended. The operations are similar in the three models: access or set a field in a tuple, segment or record; traverse a relation, tree or list in some order; select a record from a structured data item based on a Boolean condition. In addition, the relational model supports operations on entire relations, such as project and join. Even so, the set of operators can't be augmented within the database system.

A final point: Database systems don't separate cleanly type definition from data declaration. For example, in relational systems, it is seldom possible to define a relation type (scheme) independently of declaring a relation to be of that type. Thus, redundant specification is necessary to declare several relations as instances of that type.

Some design goals of GemStone are to support arbitrary levels of data structuring, to allow definitions of operations on types, and to uniformly separate type definition from type instantiation.

B. Artificial Restrictions: Database systems abound with restrictions on legal database schemes and databases. The limitations often arise from implementation artifacts creeping into the data model. Examples are limits on field lengths, the number of fields in a record, number of files, number of relations in a query, number of indexable fields, number of records in a file and depth of repeating groups.

A design goal of GemStone is to avoid arbitrary limits on the sizes of schemes and data items. Only the size of secondary storage should impose size limits on data items.

C. Structural Limitations: The data structuring capabilities of current database systems do not adequately support the complexities and variations that occur in real data. Records (segments, tuples) of a given type must be identical in structure. Every record of a given type must have the same fields and a field must draw its values from the same type in each record. At best there is an allowance for null values or missing fields. Databases are less accommodating of an extra middle name or the possibility that a company car may be assigned to a department or to an individual employee. Current systems also have restrictions on how the data structuring operations may be applied. Tuples in relations are flat records of atomic values, with no repetition of fields. A set in the network model can't have owner and member records of the same type. In the hierarchical model, databases must be strict trees; segment instances can't share subsegments (except through logical pointers).

Dynamic modification of schemes is supported only in a few extant systems. Modifying a type, such as a relation scheme, requires reformatting an entire file.

GemStone design goals are to allow for variations in structured objects, to allow arbitrary data items as values, and to support modification of database schemes without database restructuring.

D. Modeling Power: Whenever data structures in a database system won't support the actual structure of information in the real world, then the form of the real-world information gets over-simplified in the database scheme, or it must be encoded into available data structures. If the structure of the real world is over-simplified, the utility and reliability of the data is compromised. For example, if a database scheme only allows for a single middle name, two people who are distinguishable by name might become indistinguishable in the database. When information is encoded, such as flattening a set-valued field into several tuples, application programs must deal with the encoding. Encoding information also means that a database needs extra integrity constraints to ensure that only legitimate encodings appear.

A data model provides *entity identity* if the data representing any entity can be referenced directly as a unit, and the entity may explicitly appear multiple places in a database without any pointer or other indirection mechanism visible to users. Lack of entity identity leads to inconvenience in modeling and proliferation of constraints. In the relational model, to reflect that two people have the same set of children requires either a relation representing named sets of children, or a rather complicated data dependency. In the first case, the indirect reference to the set is visible to users. In the second case, the set of children cannot be referenced as a unit. Entity identity also allows easier sharing of data between data items. In the relational model, two tuples for employees assigned to the same department must represent that commonality through logical pointers to the same department tuple. Logical pointing requires that the database designer find or create a key in the departments relation. Some update anomalies come about because names of entities are used for logical pointers. Consider the previous case when employee tuples use department name to indicate the link to a department. What happens when we want to change the department name? Object-based data models provide entity identity, but other data models can be adapted for entity identity. The trend in extending the relational model is adding surrogates for tuples [Cd]. The network model supports some degree of object identity, but not in an arbitrary manner. A record may belong to more than one set, but not to two instances of the same set type.

Commercial databases to date haven't supported a hierarchy of types. We can't exploit the similarities between, say, employees and managers. so as to define common operations. Type hierarchies are common in AI systems, have been suggested for data modeling by Smith and Smith [SS] and others, and provide the organizational framework in the TAXIS data model [MBW].

Another problem with current systems is that update commands are machine-oriented. Commands insert, delete and modify parts of data items. Such updates do not necessarily correspond to any possible change in the real world, such as changing an employee's birthdate. Changes in the state of the real world typically involve updates to several database objects. Hiring an employee could involve insertions in several relations. Furthermore, the database updates for a real-world change needn't be of the same sort. Changing the times a course meets could entail both insertions and deletions in a database. Being able to model real-world changes is a powerful capability for a database system. It can help in choosing implementations for data structures, and reduce the overhead in integrity checking, since updates can be made to preserve constraints. Applications are easier to write, because common operations on the database can be expressed succinctly. In those cases where some encoding of information is needed, say for time or space efficiency, the encoding and decoding can be hidden in the update operations.

A GemStone design goal is to provide powerful data modeling capabilities through both flexible data structuring and the ability to model real-world changes.

E. Access to Past States of a Database: A temporal extension to a data model provides historical access for users and an error recovery mechanism [Cp]. Although historical access is common in manual systems, it is usually not provided in automated database systems.

People are not only interested in the current state of systems. They are also interested in the events and trends that led to a particular state, to help decide what to do next. For example, accounting, legal, financial, manufacturing and engineering applications keep and use history for auditing, trend analysis, patent applications and management tracking. For databases to be useful as models for these systems, they must capture and provide access to history as well as current state. Temporal extensions of data models are a hot research topic of late, but the results of that research haven't reached commercial systems yet, except in the form of time series packages.

Deletion was invented as a means of reusing expensive on-line computer storage. However, the cost of on-line mass storage has been rapidly decreasing. Furthermore, several emerging technologies, such as write-once optical disk [Mi], vertical recording magnetic disks [El1] and rewritable magneto-optical disks [El2] promise a significant reduction in cost per bit of on-line storage. A temporal data model replaces deletion by maintaining object history, thereby exploiting this cost trend by offering historical access for users.

Most database systems do keep a history in the form of checkpoints and recovery logs, usually stored on removable media for error recovery. However, they provide no convenient or efficient way for users to access history. A temporal data model provides both historical access and error recovery.

A GemStone design goal is to explicitly capture the history of database states as part of the data model, and to support queries that access past states.

F. Separation of Languages: In the past, computer systems have placed more emphasis on programs than data. That focus manifests itself in the design of traditional programming languages, where data that exists for the life of the program (variables) is treated differently from data that persists after execution (files). Files generally provide for much less data structure than program variables, requiring user-generated encodings for structured values written to files. In the database world, data manipulation languages do not support arbitrary computations on database objects, necessitating an interface to a general-purpose programming language. One language must be embedded in the other, either the programming language in the database language, as in PRTV [To], or the more frequent situation of calls to the database system from the programming language.

The problem with having two languages is "impedance mismatch." One mismatch is conceptual--the data language and the programming languages might support widely different programming paradigms. One could be a declarative language while the other is strictly procedural. Sometimes even a third language is involved, the operating system command language, which further complicates database interaction. The other mismatch is structural--the languages don't support the same data types, so some structure is reflected back at the interface. For example, we can access a relational database using SQL from COBOL, but when the time comes to do some computation, COBOL can only operate at the tuple level. The relational structure is lost. The reflection problem is particularly severe for entity-based data models, where identity can be lost if an entity must be mapped to a character string, a record or whatever to be passed to an application program.

A GemStone design goal is to have a single language for data manipulation, general computation and system commands.

3. Our Strategy

Our task at Servio Logic was to design a database system that overcomes the limitations listed above. We wanted a more flexible data model, embedded in a general-purpose programming language. We began with a set-theoretic data model, designed a calculus and algebra for it, and developed an algorithm to translate queries from algebra to calculus. We chose Pascal as the point of departure for the OPAL programming language. That choice proved unsatisfactory. Pascal data type definitions do not support adding operations to a type, nor hierarchies of types. Also, the Pascal type definition paradigm is rigid about typing of subparts of structures, and does not support entity identity well. Multiple use of a data item only takes place through logical references, as in the relational model, or through explicit pointers, which clashed with GemStone design goals. In addition, we saw no clean way to add system commands to Pascal.

We wanted types to have a more operational semantics, and to support entity identity better. In trying to modify Pascal to accommodate our goals, we ended up with something resembling an object-oriented programming language. The resulting data language didn't look much like Pascal at all. We knew there would be user reluctance to switch to a database system that required learning a language dissimilar to any common programming language. We scrapped the Pascal-based version of OPAL, and to begin anew with an object-oriented language, Smalltalk-80 (ST80), as a basis. While ST80 does not have a wide following of users yet, we felt it was the direction of data languages of the future, and there was enough existing literature to acquaint potential users with the concepts of the language.

4. Smalltalk-80

4.1. A Short Description

ST80 is based on three concepts: *object*, *message*, and *class*. An object is essentially private memory with a public interface. The private memory is structured as a list of named or numbered *instance variables*. An object accepts messages that ask it to access, modify or return a portion of its private memory. So that each object does not have to carry around a list of messages it handles, objects are organized into classes. A class is a group of structurally similar objects that respond to the same set of messages. The class definition contains the procedures (*methods*) that its objects use to respond to messages. Classes are organized in a (strict) hierarchy, so that they can share common structure and methods in a superclass.

For example, an object of class **ArrayedCollection** has messages to put a value at a certain index, return the value at a certain index, create a textual display of its entire value, and "grow" itself to accommodate more values. We can define a class

Employee, with each instance having a name, a set of departments and a salary. Instances might accept messages to return the employee's name, change the employee's salary, or assign the employee to different departments. A subclass **Manager** of class **Employee** could define additional structure, such as the department managed, and additional messages, such as one to return a list of subordinate employees.

4.2. What it Brings

ST80 has a simple and elegant paradigm for general-purpose programming that meshed well with our data model. ST80 objects gave us entity identity; a single object can be a component of several other objects. ST80 also provides a clear distinction between identity and structural equivalence of entities. Two entities are identical if they are represented by the same object. Two entities can have equivalent structures (have all component values the same), but not be the same object. Thus, we can distinguish, say, two gates in a circuit that have all the same characteristics, but are not physically the same gate. While this identity-equivalence distinction is not present explicitly in other commercially used models, users should recognize it when modeling their enterprises. The distinction is most obvious during update, where if two objects share a component, updates to that component through one object are visible in the other object.

The class mechanism of ST80 handled our requirements for type definition. Messages and methods give us a means to define types with operations, and to hide encodings. The class mechanism provides a hierarchy of types, albeit a strict hierarchy, so that similar types an share operations. Classes give a means of controlling the updates that can be performed on a data object, through a class's message protocol. Also, classes make database schemes easy to modify, and new variants can be constructed easily with the subclass mechanism.

Finally, we note that ST80 treats system components as full-fledged objects, giving a natural and uniform way to issue system commands from within the ST80 language.

4.3. What it Lacks

The ST80 language met our needs for type definition and general computation within a data language. Why couldn't we take the ST80 system as is and use it as the basis of a database system? The first problem is that ST80 does not handle large numbers of data items or large data items. Only 32K objects are allowed in most implementations, and the maximum size for an object is 64K bytes. We need to handle more and larger data items for the databases we want to support, such as long documents and graphical images. There are also several arbitrary size limits in the ST80 system, such as the number of variables in a method, which could be exceeded in applications we anticipate. Being a single-user system, ST80 lacks the amenities of a production database system: concurrent access of data by multiple users, recovery mechanisms, and database administrator control over replication, authorization and auxiliary structures. Because ST80 is a strictly procedural language, it does not have any declarative constructs for data manipulation. We feel that associative access to subparts of an object is a necessary aid to application program design. Also, a declarative semantics allows more flexibility in evaluating queries, and that flexibility is needed to support reasonable optimization on queries involving large amounts of data.

The ST80 model of data, while meeting many of our requirements, does not have everything we want. It does not support object histories--past values of instance variables are not retained. We want optional instance variables, without a storage penalty in instances not containing the optional variables, and the ability to add new variables to existing instances, which ST80 does not provide. We also want a path syntax for navigating through objects, and to allow assignments to path expressions. That feature allows a user to circumvent the class protocol for updating objects, but sometimes it is the most natural way to define methods.

5. Enhancing Smalltalk as a Database Language

Our first concern in designing the GemStone system is to provide a database system with more modeling power and flexibility than is available currently in commercial systems. We actually developed a data model for GemStone before beginning the design of the OPAL language. We will first introduce the Set Theoretic Data Model (STDM), and discuss its advantages over previous models. STDM turned out to be a reasonable approximation of the object structure of ST80. The structure of all the classes provided with ST80 is too complex to provide the basis for a useful data model. STDM takes a more uniform view of objects, treating most as labeled sets, and that treatment allows a fairly simple path syntax and set calculus.

We then turn to a temporal extension of STDM, and indicate how histories of object states are structured and accessed. Finally in this section, we consider the merger of ST80 and the STDM, resulting in the OPAL language and a slightly modified model, the GemStone Data Model.

5.1. A Short Description of STDM STDM is based on labeled sets of heterogeneous values, which themselves can be sets or simple values. The model builds on the work of Childs [Chi], but is not identical to his model. A piece of an STDM database might look like

```
Acme: {Departments: {A12: {Name: 'Sales', Managers: {'Nathen', 'Roberts'},
                                Budget: 142,000},
                     {A16: {Name: 'Research', Managers: {'Carter'},
                                Budget: 256,500},
            ...},
       Employees: {E62: {Name: {First: 'Ellen', Last: 'Burns'}
                                Salary: 24,650, Depts: {'Marketing'}},
                   E83: {Name: {First: 'Robert', Last: 'Peters'},
                                Salary: 24,000, Depts: {'Sales','Planning'},
                                Phones:{3949, 3882}},
            ...}}
```

(Examples in this section are not necessarily expressed in OPAL syntax.) STDM has *simple types*, generally subsets of number or character types, and *sets*. A set (denoted with {...}) has *elements*, each of which has an *element name* that labels the element and a *value*, which can be from a simple type or a set. In the sixth line of the database segment above, Name: {First: 'Ellen', Last: 'Burns'} is an element with element name **Name** and {First: 'Ellen', Last: 'Burns'} as the value. Essentially, elements function as identifier-value pairs. No two elements in a set may have the same element name.

For sets without labels, arbitrary aliases are used as element names. Presumably, the database system can generate unique aliases upon demand. In the database fragment above, the labels for department and employee sets (A12, A16,..., E62, E83,...) are aliases. In the example, we have elided element names for sets of simple values, such as {'Nathen', 'Roberts'}.

STDM uses a path syntax for accessing subparts of a set. If X is a variable whose value is the set above, then sample path expressions are X!Departments!A16!Managers and X!Employees!E62!Name.

We have developed a set-calculus query system for the STDM. As an example query, suppose we want to know employees and managers such that the employee is in the manager's department, and the employee's salary is more than 10% of the department's budget. The set-calculus expression for this query is

```
{{Emp: e, Mgr: m} where
    (e ∈ X!Employees) and
    (∃ d ∈ X!Departments) [(m ∈ d!Managers) and
        (d!Name ∈ e!Depts) and (e!Salary > 0.10 * d!Budget)]}
```

Note that a declarative syntax is needed for easy navigation through sets with aliases as element names, such as Employees. We have developed a set algebra, and an algorithm to translate a set-calculus expression to a set-algebra expression.

5.2. Advantages of STDM

STDM is a more powerful modeling mechanism than data models supported in other commercial systems. Any place a simple value can occur, we can have a set of values, or a structured value. There is unlimited nesting of sets, so a single value can have arbitrarily detailed internal structure. STDM is also flexible: optional element names in a set are handled, and elements with new element names can be added to a set. The value associated with a particular element name is not restricted to a single type. For sets representing company cars, the element name AssignedTo could have a value that is an employee, a department or a set of departments.

The power and the flexibility of STDM allows closer and more direct modeling of real-world domains. Many standard data structures can be represented straightforwardly with STDM sets. Arrays may be represented by sets with numbers as element names:

{1: {'Anders', 'Roberts'},
2: {'Roberts', 'Ching'},
3: {'Albrecht', 'Ching'}}

The index set for an array need not be positive integers—with a little more set structure it could be an arbitrary type. Record structures, with arbitrary levels and types of subrecords, are represented naturally in STDM. A relation is represented as a set of tuples, where each tuple is a set with element names corresponding to attributes of the relation. For example, the relation

A	B	C
1	3	4
1	5	4

is represented by the set

{T1: {A: 1, B: 3, C: 4},
T2: {A: 1, B: 5, C: 4}}.

Data structured as in the hierarchical model have a direct representation in STDM, by modeling a segment as a set, with elements that are field values or sets of child segments. STDM can model arbitrary combinations of arrays, sets and records.

Some structures, handled readily in STDM, would require significant encoding to fit in the relational or hierarchical models. A set-valued attribute, such as children of an employee, has to be "flattened" into several tuples in the relational model, one per child. For example, the set structure

{Name: {First: 'Robert', Last: 'Peters'},
Children: {'Olivia', 'Dale', 'Paul'}}

comes out as a three-tuple relation:

FirstName	LastName	Child
Robert	Peters	Olivia
Robert	Peters	Dale
Robert	Peters	Paul

However, under such a representation, the set of children does not exist anywhere as a single object. An alternative would be to form an additional relation to represent sets of children. This alternative requires names for the sets of children, and introduces an artifact at the user level, requiring extra joins to bring the description of an employee together. There is unavoidable redundancy in either approach: Some value is going to be repeated three times. Operations on the set, other than insertions and deletions, are awkward to express. For example, stipulating one set is the subset of another set requires two quantifiers in relational calculus. STDM, while requiring some encoding for certain data items, at least preserves the integrity of entities. A set of children can be represented and manipulated as a single data item at some level. Reducing the amount of encoding to model real-world entities, and being able to preserve the integrity of those entities, makes data easier to understand and applications simpler to write.

The declarative syntax for STDM simplifies many application programs. It also allows much more access planning by the database system than with an equivalent query specified procedurally. The set calculus provides the basis for a declarative query syntax. A distinguishing feature of our calculus, as compared to relational calculus, is that variables can be bound to functions of other variables, rather than only to fixed database objects. Because a variable may assume a value that is a structured object, a condition such as

$m \in$ d!Managers,

makes sense.

5.3. Adding History to Objects

5.3.1. Transaction Time vs. Event Time

The first question to face when adding time to a data model is What kind of time to incorporate? *Transaction time* is the time when an event is recorded in the database. *Event time* is the time when an event occurs in the real world. GemStone uses transaction time to record history, rather than event time. We discuss that choice.

In many databases, transaction time and event time are either the same or close enough to be considered equivalent. Furthermore, as computer systems are used to more completely automate our real-world systems, databases are increasingly becoming more real-time, in which the official time of an event is the moment when the event is recorded in the database. Even though a database is a model of a real-world system, the database itself is a real-world system. Thus, the recording process itself is a significant real-world event whose event time is its transaction time.

Transaction time has a simple semantics that is application independent. Thus, transaction time is easily automated by a database system. In contrast, the semantics of event time is application dependent, so that event time is difficult for a database system to automate on behalf of users. Note that the use of transaction time to record history in GemStone does not imply that event time cannot be captured as user-defined data. In fact, the extendibility of classes that OPAL provides allows any semantics for time to easily be added by users. Since transaction time is system-generated, and cannot be modified by users, it provides high integrity. In contrast, event time must be modifiable by users when a discrepancy is discovered between the real world and its database model.

When we venture to extend the scope of computer science, we often have no empirical basis within our young field upon which to base our decisions. In the case of extending a data model to manage history, we do have some empirical basis in the fields of accounting and other recordkeeping disciplines. Accountants have dealt with the issue of transaction time vs. event time for many years, if not centuries. They have seen as fundamental the consideration of the recording process itself as a significant real-world event. Reed [Re1, Re2] argues that storing transaction time is useful for synchronizing concurrent transactions. This observation offers the possibility of sharing the overhead of generating and storing the transaction time over both functions.

5.3.2. The Temporal Extension

In the data model presented so far, we represent each element of an object as an element name-value pair. We represent history in STDM by replacing an element's single value with a set of values. The element name is associated with each value via a transaction time. The binding between an element name and its associated value is indexed by time. For example, if E is a set representing an employee, then E!Salary is actually a mapping from transaction times to values giving the salaries of the employee in past and present states of the database. If T represents a point in time, then E!Salary@T gives the value of E's salary at time T. That is, E!Salary@T is the value that E!Salary had in the state of the database that existed at time T. The relationship between an element name and a particular value

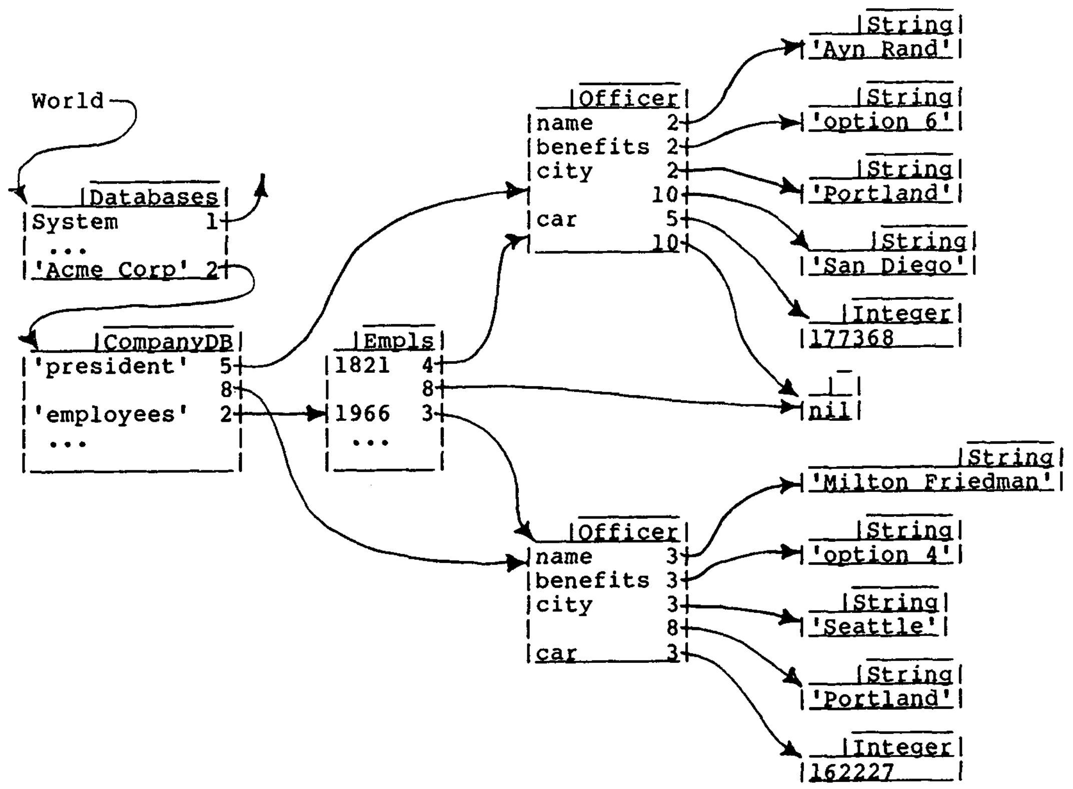

Figure 1: A Database with History

begins at the transaction time for the value, and ends when a new value is associated with the same element and a later transaction time. Objects themselves do not have time. Only their relationships with their elements are indexed by time.

The example in Figure 1 illustrates the temporal extension. The numbers at the right side of each box represent the transaction times of relationships. The example shows that at time 8, the company president was changed from 'Ayn Rand' to 'Milton Friedman'. The example represents a history of two presidents. Ayn was president from time 5 to 8. Milton was president from time 8 to the current time. Prior to this change, Milton had worked for the company in Seattle. His new appointment required a move to Portland. The example represents a history of Ayn as an employee from 2 to 8. The fact that Ayn left as an employee is indicated by the relationship in the employees object with her employee number 1821 as an element name and with time 8, whose value is the object nil. Shortly after leaving the company, Ayn moved to San Diego. She was allowed to continue to use her company car until her move at time 10.

A current transaction can access the new company president by the path expression

World!'Acme Corp'!'president'

or at a time in the recent past with the path expression

World!'Acme Corp'!'president'@10.

If the argument of @ were 7, then the previous president would be accessed. As another example, the previous president's current city, San Diego, can be accessed by the path

World!'Acme Corp'!'president'@7!city.

5.4. Merging ST80 and STDM

As we said, STDM was conceived before ST80 was chosen as the basis for OPAL. STDM provides the basis to add path syntax, set calculus and object history to ST80 to get OPAL. Conversely, there some deficiencies in STDM that ST80 helped overcome. While STDM is powerful and flexible, it did not provide all the modeling power we wanted. STDM does not support entity identity, except for simple, non-changeable values. STDM sets are unlike mathematical sets, in that any set instance can be an element in at most one other set. In mathematical sets, of course, an element may be a member of several sets. If two sets represent two employees in the same department, they could have logical pointer to the same department set, or they could duplicate the department set as an element. Neither alternative is entirely satisfactory. The first requires navigating across data objects at the logical level; the second entails added complexity when updating a department description. Furthermore, without entity identity, STDM cannot capture network schemes directly, where a single record may be an element of several sets. STDM does not have a good type definition mechanism. We had a hard time deciding whether types should be associated with sets or elements, or both; how to describe types; how to specify operations on instances of a type; and how to factor element names from instances of a type. Also, we had no structure on types--there was no type hierarchy.

Definition of operations for a type was especially important to us, as certain real-world structures still required encoding. Thus, we needed a mechanism for information hiding, to make database objects appear and behave as their real-world counterparts. For example, to make a relation behave as a relation, we needed a tuple to appear as a mapping rather than a set. We also wanted to define subclasses of simple types with operators and orderings different from the parent type.

STDM does not provide a general programming interface. Set calculus is its only vehicle for describing set access and update. We needed a procedural language to support the set algebra. Further, we found many examples where procedural updates would be clearer and more succinct than the calculus equivalents. We also wanted to include general computations in the conditions of calculus expressions.

Merging STDM with ST80 to overcome these deficiencies is straightforward. We can identify sets and simple values in STDM with objects in ST80 and elements with instance variable-value pairs. Just as STDM induced changes in ST80 to get OPAL, ST80 dictated changes in STDM. We call the revised model the GemStone Data Model (GSDM). In STDM we had concentrated on names--path names and element names—in trying to assign types. In GSDM the focus became values (objects), and that is where typing went on. We still feel that some typing of element names could give us big performance advantages, such as more flexibility on access paths when processing declarative queries. and we are looking at this extension to OPAL, as are others [BI, Ha].

The switch from Pascal to ST80 as a basis for OPAL was propitious. We have a single language for general programming, data manipulation and system commands, with excellent support for data and operation definition. There is no impedance mismatch between application language and database language, which we noted is particularly troublesome for a database that tries to support entity identity. GSDM has true entity identity, not just entity integrity. GSDM retains the time dimension, path expressions and set calculus from STDM. We have been able to incorporate declarative statements in OPAL without departing from Smalltalk syntax. Thus, our realization of set calculus is particularly powerful, as it can *include* procedural parts, and can be *included in* procedural methods.

We have been careful to preserve entity identity in GSDM as regards time. Although entity identity is important in databases that do not model history, it is even more important when object history is modeled. Identity is a property of an object that spans time. When an object is instantiated, it is given a globally unique identity. It lives forever with that identity, although the value of its elements may change, and it may not be accessible from other objects in some states of the database. In OPAL, we have eschewed the !-@ notation for navigating through object histories in favor of a *time dial*. We feel that almost all navigation through history would be within a single past state of the database. Setting the time dial to time T is the same as appending @T to each component in a path expression. A useful feature of the time dial is the system variable SafeTime. A read-only transaction can set its time dial to SafeTime to get the most recent state for which no currently running transaction can make changes.

Support for views drops out almost for free. We can construct an object that provides a view, and that object can employ other objects, procedural statements and calculus expressions to define the extension of the view. Furthermore, since the view object can retain connections to the objects that contributed to the view, and since it can support its own methods for messages, view updates are more manageable than in other data models.

6. Enhancing Smalltalk as a Database System

The language extensions covered in the last section do not make ST80 into a database system. The problems of supporting concurrent access to large amounts of data, along with replication and recovery, remained to be solved. We have added classes and primitive methods to OPAL to provide transaction control, storage hints and requests for replication of data. Of course, the biggest requirement of the GemStone system is that it manage objects with all these database features efficiently. In this section we briefly cover the current implementation of GemStone.

The GemStone system is currently being implemented on special-purpose hardware, although we anticipate purely software versions to run on other vendors' hardware. We expect to obtain efficiency by having the database system control secondary storage directly, without an intervening operating system. A separate operating system would be somewhat oblivious to the storage needs and access patterns of the database system. Also, an independent operating system can play havoc with requirements on database consistency, security and recoverability. Disk access will always be by entire tracks, as a track is the natural unit of physical access for a disk. Also, by having a declarative query language, we have the latitude in processing queries to exploit fully secondary storage layout, directories, and special hardware.

Communication with GemStone is done in blocks of OPAL source code. Compilation and execution of those blocks is done entirely in the GemStone system.

The system structure of GemStone is similar to that of ST80, minus display and file system classes, but with additions for set calculus, path syntax, time, concurrency, authorization, recovery, replication and directories. The display part of ST80 is not currently needed, as our present implementation has GemStone running on its own hardware and communicating to user interface programs on host machines through a network link. We do envisage a stand-alone GemStone system eventually, which would support display control. All file system duties are assumed by the database system.

The GemStone system has two main parts, the *Executor* and the *Object Manager*. The Executor is responsible for controlling sessions in the GemStone system on behalf of users on host machines. The Executor handles communications between GemStone and host software: receiving blocks of code, returning results and error messages. It maintains a *Compiler* and *Interpreter* for each active user. The Interpreter is an abstract stack machine that executes *compiledMethods* consisting of sequences of *bytecodes*, much the same as the ST80 interpreter. It dispatches bytecodes, performs stack manipulations and some primitive methods, and makes calls to the Object Manager. The Compiler requires some modifications from the ST80 compiler. Most are small changes in syntax or for slightly different bytecodes, but a large addition is needed translate calculus expressions into procedural form.

The Object Manager performs the same operations as the ST80 object memory, but is quite different in structure. The Object Manager also handles operations related to concurrency and secondary storage management: transaction control, authorization, data replication, recovery and directory management. In addition, the Object Manager responds to messages to conduct its fetches in some previous state of the database. Each user session in the GemStone system has its own invocation of the Interpreter, and its own Object Manager with a private object space. Sessions have shared access to the permanent database through transactions.

In the standard implementation of the ST80 object memory, objects are represented as blocks of contiguous words of memory, with ***object-oriented pointers*** (OOPs) to the values of instance variables. Since GemStone objects retain history, they grow with time, and a fixed block of memory is not a feasible representation. In the GemStone Object Manager, the implementation of objects is based upon ***associations***. An element is represented as an element name and a table of associations. The associations are pairs of transaction times and object pointers, each representing that the element acquired the object as its value at the time given by the transaction time. The mapping from arbitrary times to value for an element can easily be realized from this table. Objects are broken into elements and associations, which are organized into a linked list under a header for the object. A directory may be interposed between the object header and the participating elements. Such a directory is useful when an object has a long history, or it represents a set without labels, whose elements will likely be accessed associatively, rather than through element names. Between objects, pointers to elements are usually physical pointers, as we expect most of the data to be strict tree structures. Thus, physical access paths parallel logical access where objects aren't shared. Where an object is an element of more than one set, one logical path is chosen as the basis for the physical access path, and other references to the object use a ***global object-oriented pointer*** (GOOP). The GOOP is resolved through a ***global object table*** to get the primary logical path to the object, from which its physical access path can be deduced.

The Object Manager has several major subcomponents. The *Transaction Manager* is shared by all invocations of the Object Manager, and handles concurrent use of the permanent database in an optimistic manner. It records accesses to the database for each session, and validates them for consistency when a transaction commits. The *Directory Manager* creates and maintains directories. Directories use standard techniques modified to handle object histories. One headache has been that hints given in OPAL for structuring directories must be translated for use by the Object Manager. Another problem is using a nested element as a discriminator. Since that element may be different in different states of the database, its object may need to appear along two branches of the directory.

The *Linker* incorporates updates made by a transaction in the permanent database at commit time, calling for restructuring of directories as needed. The Linker is called by the *Boxer*, whose job it is to fit objects into tracks after database changes. The *Track Manager* schedules reads and writes of tracks. The *Commit Manager* provides *safe writing* for groups of tracks. Safe writing guarantees that all the tracks in the group get written, or none get written, and that the tracks in the group replace their old versions atomically.

The time dimension of GSDM leads to one simplification in the GemStone Object Manager over ST80 object memory. Database objects in the past never go away, as references exist to them in some state of the database, and, theoretically, all past states of a database are preserved. Thus, no garbage collection need be done on database objects. Temporary objects created by user sessions may have to be garbage collected. However, again the task is eased, as an entire session workspace can be discarded at the end of a session. A database administrator can explicitly move objects to other media, such as tape or write-only memory. Hence, while conceptually the entire history of the database exists, some objects in it may become temporarily or permanently inaccessible.

7. Similar Approaches

Others have attempted to make ST80 a better vehicle for database applications. Most of those efforts concentrate on supporting data in secondary storage. Probably the best-known extension to secondary storage is the Large Object-Oriented Memory (LOOM) [KK]. LOOM maintains a two-level object space in main memory and on disk. Objects are moved to main memory from disk as needed. LOOM does not meet our needs for four reasons. First, it is intended for a single user system. Second, while it allows many more objects than standard Smalltalk implementations, it retains the same maximum size for objects. Third, it uses the standard Smalltalk representation of objects. That representation is not suitable for us, as we have objects with a time dimension. For objects with a large history, we may want to bring only a fragment of the object into memory. Fourth, LOOM hasn't completely dealt with the problems of clustering and indexing in secondary storage.

Several recent data models undertake to solve many of the same problems that GSDM does. The Cypres database system [Ca] supports an entity-relationship-datum data model with a strong notion of object identity. However, Cypres does not provide general-purpose programming, nor operations definition for types. Morgenstern [Mo] describes the data model for the IM project at ISI. That model is also entity-based, and it provides a hierarchy of data types, but again, it doesn't provide operation definition. IM does address integrity constraints in greater depth than we have yet.

Programming language-data language integration has been tried in combinations other than a set-theoretic data model with an object-oriented language. Some of the first attempts were adding relations to highly-typed languages such as Pascal. Two examples are Pascal/R [Sch] and PLAIN [W+]. IBM has at least two efforts in the area. One is the EAS-E system [MMP], which provides a network data model in a structured-English programming language. The other is the IDE project at the Los Angeles Scientific Center [Ne], which uses an object-based data model. ADAplex [C+] combines the Daplex functional data model with the object-based language Ada. Beech [Be] describes IDL, which has an object-based data model, with a Formula type for defining operations on classes of objects. Finally, the combination of a relational database system with a logic programming language, such as Prolog, has gathered many adherents [Da, Pa1, Pa2, Wa].

8. Acknowledgements

Fred Boals did the initial work on the set calculus to set algebra translation algorithm, and Bob Johnson brought it to its current form. The adaptation of OPAL from ST80 was done for the most part by Alan Purdy and Jason Penney. The design of the Object Manager is due to Dave Schrader, Bob Bretl, Allen Otis, Walt Gorman, Pat Timlick and Lynn Nelson. Most of these people read earlier versions of this paper, and while the authors should accept responsibility for remaining errors themselves, they prefer to ascribe blame to the others.

9. Bibliography

[Be] D. Beech Introducing the integrated data model. Hewlett-Packard Computer Science Laboratory technical note CSL-15, January 1983. 2.

[Bl] A.H. Borning, D.H.H. Ingalls. A type declaration and inference system for Smalltalk. Univ. of Washington Computer Science TR 81-08-02a, November 1981.

[Ca] R.G.G. Cattell. Design and implementation of a relationship-entity-datum data model. Xerox PARC report CSL-83-4, May 1983.

[C+] A. Chan, et al. DDM: An Ada-compatible distributed database manager. Digest of Papers, COMPCON '83, February-March 1983, 422-425.

[Chi] D.L. Childs. Feasibility of a set-theoretic data structure based on a reconstituted definition of relation. IFIP '68, North-Holland, 1969, 162-172.

[Cd] E.F. Codd. Extending the database relational model to capture more meaning. *ACM TODS 4*, 4, December 1978, 397-434.

[Cp] G. Copeland. What if mass storage were free? *IEEE Computer 15*, 7, July 1982.

[Da] V. Dahl. On database systems development through logic. *ACM TODS 7* 1, March 1982, 102-123.

[DE] *Database Engineering 6*, 4, December 1983: issue on expert systems and database systems.

[El1] *Electronics 56*, 8. Perpendicular bits up density of prototype disk drives. April 1983.

[El2] *Electronics 56*, 14. Erasable optical disk system could be available by 1985. July 1983.

[GR] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Ha] R. Hagmann. Preferred classes: a proposal for faster Smalltalk-80 execution. In *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Ed., Addison-Wesley, 1983, 323-330.

[KK] T. Kaehler, T. Krasner. LOOM--Large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Ed., Addison-Wesley, 1983, 251-272.

[MMP] A. Malhotra, H.M. Markowitz, D.P. Pazel. EAS-E: An integrated approach to application development. *ACM TODS 8*, 4, December 1983, 515-542.

[Mi] S.W. Miller, editor. Special issue on mass storage systems. *IEEE Computer 15*, 7, July 1982.

[Mo] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. VLDB IX, Florence, Italy, October 1983.

[MBW] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong. A language facility for designing database-intensive applications. *ACM TODS 5*, 2, June 1980, 185-207.

[Ne] P. Newman. Techniques for environment integration. Computer Science colloquium, Oregon Graduate Center, October 1983.

[Pa1] K. Parsaye. Database management, knowledge base management and expert system development in Prolog. ACM SIGMOD Database Week, Databases for Business and Office Applications, San Jose, May 1983, 159-178.

[Pa2] K. Parsaye. Logic programming in relational databases. In [DE], 20-29.

[PP] Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling, Pingree Park, Colorado. SIGMOD Record 11, 2, February 1981.

[Sch] J.W. Schmidt. Some high level constructs of data type relation. *ACM TODS 2*, 3, September 1977, 247-261.

[SS] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM TODS 2*, 2, June 1977, 105-133.

[To] S.P.J. Todd. The Peterlee Relational Test Vehicle—a system overview. *IBM Systems Journal 15*, 4, December 1976, 285-308.

[Wa] D.H.D. Warren. Efficient processing of interactive relational queries expressed in logic. VLDB VII, Cannes, France, September 1981, 272-281.

[W+] A.I. Wasserman, et al. The data management facility of PLAIN. ACM SIGPLAN Notices 16, 5, May 1981, 59-80.