A Performance Comparison of Object and Relational Databases Using the Sun Benchmark

Joshua Duhl & Craig Damon

Ontologic, Inc.
47 Manning Road
Billerica, MA., 01821

Abstract

A general concern about object-oriented systems has been whether or not they are able to meet the performance demands required to be useful for the development of significant production software systems. Attempts to evaluate this assertion have been hampered by a lack of meaningful performance benchmarks that compare database operations across different kinds of databases.

In this paper, we utilize the Sun Benchmark [Rube87] as a means for assessing the performance of an object database and comparing it with existing relational systems. We discuss the benchmark, and many of the implementation issues involved in introducing a relationally oriented benchmark into an object-oriented paradigm. We demonstrate the performance of an object database using Ontologic's Vbase object database platform as an example of a commercially available object database, and we compare these benchmark results against those of existing relational database systems. The results offer strong evidence that object databases are capable of performing as well as, or better than, existing relational database systems.

1. Introduction

The promise of object-oriented databases has been that they can potentially provide faster performance than traditional database management systems. This promise has made by many people, and while not yet verified, it has often been a focus of discussion. On the one hand, proponents state that object databases will provide higher performance for several reasons: Operations can be performed on individual objects or classes of objects; Sub-components can refer to an object by object identity rather than by state (key); Object references can be cached for in-memory access times; and complex design components can be represented more directly using objects than with relational systems [Maie86]. On the other hand, higher levels of abstraction generally lead to worse performance. Analogously, one would expect a high level language, such as PL/1, to be slower than a lower level language, like Fortran, and Fortran to be slower than assembler. One of the major attractions of object systems is the high conceptual level and abstraction at which users can approach, interact with, and model their problem domains. Implementing this high level of abstraction, following this vein, would lead to poor performance because the overhead incurred in supporting the abstraction model will be too great.

Object databases are an emerging and developing technology. Until last year, object databases existed either in early stages of development or as research prototypes. As such they were basically unsuitable for fair performance comparisons with established relational systems. In the past year, however, several object databases have reached a plateau in their development where they can be deemed suitable for performance comparisons with existing relational systems.

Several benchmarks have been developed for measuring database performance. The common database benchmarks, such as the Wisconsin benchmark [Bitt83] and the TP1 benchmark [Anon85] are closely targeted at relational database usages, which emphasize operations specific to the relational model and high volume transaction processing usages respectively. These are inappropriate measures for the kinds of comparisons of interest to users of object-oriented systems. The Sun benchmark was developed in an attempt to measure fundamental database operations in a usage pattern more typical of engineering applications. This benchmark more closely approaches the kind of benchmark suitable

for comparison of object and relational database systems.

2. The Sun Benchmark

The Sun Benchmark was developed to establish an acceptable metric for measuring the *response time* of simple database operations. While it is strongly influenced by the relational model, it is intended for use across different kinds of database systems, and attempts to provide generic, data model independent definitions and benchmark tests that are suitable for benchmarking databases of any data model.

The benchmark itself consists of a simple schema and a representative set of fundamental database operations. Each of the seven operations is measured in terms of its *response time*, the elapsed time between the invocation and return of the operation. The Sun Benchmark is summarized below. Portions of the summary are taken directly from [Rube87].

The schema consists of 3 *record types*: a person record, a document record and an author record. A *record type* is defined as a group of *records* with the same field types. It is a relation in a relational system and a class or type in an object-oriented system. A *record* is defined as a set of fields. It corresponds to a tuple in a relational system and an object in an object-oriented system. Often the fields of an object may be group or multiple valued. The Sun Benchmark definition allows for this specific variation between an object and a record. It also defines a *key* as a unique field over all records of a type. Examples would be a relational primary key or a unique object identifier (UID).

Each of the three record types consists of a set of fields:

1. A **Person** has three fields: a person ID, a name and a birthdate. All IDs are 4 byte integers, serving as keys. Birthdates are randomly generated 4 byte integers that span a 100 year interval. Names are randomly generated strings of up to 40 bytes.
2. A **Document** has seven fields: a document ID, a title, a page count, a document type, a publication date, a publisher, and a description. Titles, publishers, and descriptions are strings up to 80 bytes. Page count, document type and publication date are integers.
3. An **Author** has two fields: a person ID or key referencing a person and a document ID or key referencing a document. It connects each person to zero or more randomly selected documents and each document to exactly 3 randomly selected people.

The Benchmark specifies constructing two versions of the database: The small database is populated with 20,000 Persons, 15,000 Authors and 5,000 Documents. The large database is a factor of ten larger than the small version (i.e. 200,000 Persons, etc.).

The benchmark is comprised of seven individual benchmark operations, defined as follows:

Name Lookup: Fetch the name of a person with a randomly generated ID.

Range Lookup: Fetch the names of all people with birthdates within a particular randomly generated 10 day range.

Group Lookup: Fetch the author ID's for a given random document ID.

Reference Lookup: Fetch the name and birthdate of a person referenced by a randomly selected author record.

Record Insert: Store a new author record.

Sequential Scan: Serially fetch records from the document table, fetching the title from each, but without performing any pattern match computation on the title.

Database Open: Perform all operations necessary to open files, database schema information, and other data structures and overhead to execute the benchmarks, but not time to load the application program itself.

Each operation is performed 50 times, and the entire set, except for database open, is repeated 10 times to simulate an entire session with an engineering application with a mix of different database operations. The reported results are therefore averaged over the entire 500 iterations of each benchmark operation, or 50 iterations in the case of database open.

3. Overview of Vbase

Vbase [Andr87, Onto87] is the object-oriented database platform developed by Ontologic, Inc. It combines the characteristics of object-oriented

languages, as set forth by [Wegn87], i.e., objects, classes (types) and inheritance, with the common DBMS characteristics of persistent storage, i.e., multi-user support, concurrency control, and transactions. Vbase incorporates the notions of data abstraction and strong type checking, notions which are both heavily influenced by the CLU programming language [Lisk81].

Abstract types are specified using the declarative Type Definition Language (TDL). A TDL definition specifies the abstract behavior of an object or class of objects. The behavior of an object, its state (properties) and the set of operations defined on that state, is defined by the object's abstract type, or inherited from the object's supertype. Vbase supports a large hierarchy of kernel types, including a sub-hierarchy of 13 aggregate types. Aggregate types include arrays, dictionaries, stacks, and sets. Vbase also supports user defined types, free operations (operations unassociated with any specific type), storage class information, and representation manager specification through TDL.

Objects are manipulated through a compiled procedural language, COP, which is an extension of the C language and is strongly influenced by CLU. COP extends C by adding iterators (operations which iterate over an aggregate of objects yielding each object in turn), exception handling, and syntactic support for object operation invocation and property access. Currently all Vbase applications are written in COP.

In Vbase the specification of an object is separate from its underlying representation. The operations specified in TDL are performed by methods implemented in COP. This distinction between specification and representation allows for changes to be made to the implementation without altering the abstract model.

4. Sun Benchmark on Vbase

The Sun Benchmark is meant to be data model independent. However, it is evident that its conceptual basis and construction are strongly influenced by the relational paradigm. As such some of the relational constructs or operations do not. correspond to equivalent object-oriented counterparts. When introducing this essentially relationally oriented benchmark into an object-oriented system, we encountered both modeling and implementation differences, some of which have been previously encountered in other similar efforts[Smit87].

4.1 Modeling Differences

One of the greatest variances of object-oriented systems from relational is the object identifier provided in an object system. In a relational system, tuple identification is provided by a fabricated unique key that must be maintained for each record. Most tuple accesses are performed using this key. In object-oriented systems, unique references (UID's) are generated as a part of object creation. They provide object identity and can be freely used at any time (across all time) with safety, independently of the field values of an object. As explicitly allowed by the benchmark definition, we have used the Vbase object reference as the key in all cases where appropriate.

A property that refers to another object usually holds the UID of object it references. As such, UID's allow for *direct connectivity* between objects. However, in Vbase, as in many object systems, an object's UID also contains information about the object's type. This differentiates simple direct connectivity, such as with pointers and as found in the network model, from what could be called typed direct connectivity. This additional type information embedded in the object reference can be used to provide information for semantic validation. Unlike relational systems, object systems inherently support this notion of direct connectivity.

Another variance from relational systems is that object-oriented systems support the notion of aggregates or aggregate objects. Aggregate objects allow a group of similarly typed objects to be referred to and manipulated as a single object. They are container objects. Their behavior may be subtyped and refined so as to support combinations of ordering, keyed access, and allowing for multiple copies of the same instance (i.e.. multisets or bags vs. sets).

A substantial variance from the relational model is the disappearance of the Author record type. In the Sun Benchmark, the Author record type exists solely to provide a connective relation between persons and documents. It serves as a many-to-many connective modeling construct. Vbase

inherently supports the notion of *distributed properties.* A distributed property is a multi-valued property, which refers directly to each individual object in the set. When used with inverses, distributed properties provide a direct implementation of a many-to-one mapping, or in the case where the inverse property itself is distributed, a many-to-many mapping. Because distributed properties can provide direct connectivity between a single object and many related objects, the need for an intermediate connective modeling construct vanishes entirely.

Conceptually, a person can be considered having authored a document if the person has in fact written a document. In an object system, this relationship can be modeled in several ways. An Author can be modeled as a subtype of type Person or it can be modeled as an optional property (relationship) on type Person (between the person and the documents authored). If Author is modeled as a subtype of type Person, then when a person writes a document and has it published he or she becomes an author. In an object system this could be modeled with dynamic type acquisition. The type of the Person object is changed and in fact specialized to Author, because the Person object has changed. Since Vbase currently does not support the notion of dynamic type acquisition, we have modeled the notion of an author as an optional property. In this way, the potentiality of becoming an author is maintained.

Another fundamental modeling difference from relational systems is found in an object system's support for operations. Relational systems generally perform well when accessing a field in all tuples of a relation but perform poorly when accessing individual records. Operations in object systems are primarily type specific, and are tailored to the type of object they operate upon, so performance is generally high. In object systems, operations also encapsulate behavior, and as such encapsulate code performing some application function that in relational systems normally resides in the application. In Vbase the abstract specification of an object is separate from its implementation. This is particularly useful for operations whose behavioral implementation (their method) can be refined and optimized without affecting the abstract specification or operation invocation.

4.2 Implementation Differences

The benchmark can be implemented in two ways. It can either use all three types and behave like the table oriented relational model, which we will refer to as the *object-relational* (OR) version, or it can use only two types and can take advantage of the direct relationships inherent in the object-oriented paradigm, which we refer to as object version.

The OR version models the original three types, Person, Author and Document. The Person and Document types link directly to the Author type, which models the intermediate table necessary for the many-to-one, one-to-many and many-to-many relationships in a relational system. Type Author has two properties, personlink and documentlink that are both optional multi-valued inverse properties. The documentlink property, denoted author$documentlink links directly to the Document$authorlink property on type Document. The Author$personlink property links to the Person$authorlink property on type Person. If either property is modified the inversely linked property is also modified automatically by Vbase. The TDL is listed below.

```
define type person
  import Document, Author;
  supertypes = {Entity};
  classtype = $ExplicitClass;
  properties = {
    name : String;
    birthdate: Integer;
       authorlink: distributed Set[Author] inverse
                Author$personlink;
  };
end Person;

define Type Document
  import Person, Author;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    title : String;
    pages : Integer;
    type  : Integer;
    date  : Integer;
    publisher : String;
    description : String;
       authorlink: distributed Set[Author] inverse
                      Author$documentlink;
  };
end document;
```

```
define type Author
  import Person, Document;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    documentlink : optional Document inverse
                    Document$authorlink;
    personlink : optional Person inverse
                    Person$authorlink;
  };
end author;
```

The object version models only two of the three types, Person and Document, taking advantage of distributed properties available in the Vbase object model to eliminate the intermediate Author type. The notion of authorship is fulfilled through an inverse relationship property. The many-to-many mapping is accomplished by a set of direct inverse links between the document's *authors* property, denoted Document$authors, and the *publications* property on Person, Person$publications.

With this combination of modeling constructs, the semantics of authorship can be modeled quite naturally: if it has a value, then the person has authored a document, and if it is empty, then the person has not authored anything. Direct, inverse, and optional properties are modeling features inherent in Vbase that also result in significant performance advantages, both in size and speed. Other than the elimination of the Author type, the object version TDL differs only slightly from the OR version:

```
define type Person
  import Document;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    name : String;
    birthdate: Integer;
    publications : distributed Set[Document] inverse
                    Document$authors;
  };
end Person;

define Type Document
  import Person;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    title : String;
    pages : Integer;
    type  : Integer;
    date  : Integer;
    publisher : String;
    description : String;
    authors : distributed Set[Person] inverse
                    Person$publications;
  };
end Document;
```

Note: The full TDL can be found in Appendix A.

In both versions, optimized implementations of the create operation (the create method) have been written to achieve faster performance. Additionally, a special iterator method has been written to yield people whose birthdays fell within the 10 day range of birthdays as required by the range lookup operation. Birthdates are stored as keys into a B-Tree holding objects of type Person. The iterator made use of a pre-release functional interface to the B-Trees that is to be integrated into the 2.0 release of Vbase.

The aggregate classes containing all People or Documents have been implemented in the small version of the benchmark as Lists. In the large version they will be implemented as B-Trees.

The group lookup is implemented as an iteration over the set of Authors on a given random document.

The name lookup is performed as a property access on the name property on the random people.

The reference lookup is performed as two property accesses on each random Person.

5. Results

5.1 Benchmark Environment

The Benchmark encourages performance optimization. It suggests taking advantage of cache memory, permitting caching of as much of the database as the database system allows, and utilizing the fastest and most efficient data structures and access methods for each benchmark operation. The Vbase cache size is user determinable with a current

Sun Benchmark Data [1]

Small (40,000 instances) Database Results Only

DBMS	INGRES	UNIFY	RAD-UNIFY	VBASE	VBASE
Model Type	Relational	Relational	Relational	Relational	Object
Name Lookup	35 ms	60 ms	9 ms	11.4 ms	9.8 ms
Range Lookup	393ms*	358 ms	76 ms	95.9 ms	84.1 ms
Group Lookup	116 ms	85 ms	24 ms	6.0 ms	5.1 ms
Reference Lookup	165 ms	50 ms	6 ms	30.6 ms	9.8 ms
Record Insert	56 ms	230 ms	43 ms	95.44 ms**	24.3 ms***
Sequential Scan	2 ms	11 ms	3 ms	1.53 ms	1.6 ms
Database Open	1300 ms	580 ms	580 ms	1020 ms	1036 ms
Actual Size	6.3 MB	3.8 MB	3.8 MB	12.1 MB****	10.4 MB#

Figure 1.

* These are estimated, not empirical times. See [Rube87] for more detail.

** This is implemented as a Create operation.

*** An alternative implementation of an insert function would involve a Create operation in Vbase. The averago time for this is 75.7 ms.

**** This figure represents 5.5 MB of kernel data, and 6.6 MB of benchmark data.

\# This figure represents 5.5 MB of kernel data, and 4.8 MB of benchmark data.

[1] The figures in this table for Ingres, Unify, and RAD-Unify are presented in [Rube87]. Original data for Vbase are added by the authors.

ONTOLOGIC, INC.
June 20, 1988

limit of about 8 megabytes. We found 6 megabytes to be optimal for the small database. At the writing of this paper, the large database benchmark has not yet been completed.

The original Sun benchmark was performed on a Sun 3/160 processor, with 8 megabytes and a local database stored on disk. Our version was executed on a Sun 3/160 processor with 16 megabytes of physical memory, and the database stored on a local disk, using version 1.1 of Vbase.

The difference in the physical memory size affected the ability to cache more of the database in main memory. With more physical memory available, a larger cache size is also possible. Thus physical memory size indirectly affects benchmark performance through the cache size. When the cache size remained constant and the benchmark was executed on machines with different sized physical memories there was no noticeable difference in the benchmark timings.

In implementing the benchmark support code, the most awkward phase is randomly generating the UID references, since they are not simply consecutive integers. Instead, we build an in-memory array containing all of the references to a given type, and randomly chose an entry in this array. To ensure that building this array does not bias the results in any sense, we close the database and flush all remnants of this activity (save these arrays) before running any of the tests. Loading this array is not included in the actual benchmark timings.

5.2 Discussion

The table in Figure 1 summarizes the results of the small database version of the Sun Benchmark, and allows for comparison of both Vbase models to the existing relational versions. The large version has yet to be completed at the time of the writing of this paper. Rad-Unify, developed by Rubenstein, et. al., is an in-memory version of Unify. It caches as much of the database as possible, which in the small benchmark is the entire database. It also utilizes a simplified locking mechanism that allows for only one writer at a time. Vbase also caches the entire database for the small benchmark, as well as providing full multi-user support.

Overall, these numbers indicate that an object system can meet and in many cases exceed the performance of a fast relational system, even in a problem clearly from the relational domain. The results also indicate that an object system can model a relational implementation and achieve response times comparable to the relational systems. Note that it is possible to achieve improvements in performance by using an alternative schema definition.

5.2.1 Trends

For several of these tests, the early timings dominated the averages, themselves being dominated by the disk transfer time of the schema objects as well as the data itself. All the tests show continual improvement in the timings over the course of the benchmark, particularly after the first 100 iterations and even more so after 200 iterations.

The name lookup test drops dramatically over the course of the benchmark coming to a constant 0.8 milliseconds (ms) after 350 iterations. The reference lookup follows a similar trend, dropping to approximately 3.2 ms after 300 iterations. This trend occurs because more of the referenced objects have been brought into the cache and can be found in the cache as the benchmark proceeds. The trend applies, but to a lesser extent, to all of the tests except the Sequential Scan test, which maintains a fairly constant timing after the first 50 iterations. It should be noted that Rad Unify exhibits similar behavior.

For the record insert test, we provided two numbers for the object version. To implement the behavior described in the benchmark using Vbase, no creation was required. Instead, an extra value was placed in the distributed (multi-valued) property on both the selected Person and the selected Document. While this is a meaningful distinction between relational and object systems (many of the tuples created in a relational system exist only to express relationships and would not be required on an object system), creates remain a necessary and important component of any database system. As the timings indicate, this is the one area where the performance of Vbase is relatively slow. This is not a commentary on object databases, but a limitation in the current implementation of Vbase.

Vbase supports a generic create method that is called to create any new entity. The original timing for the alternative record insert, using a create, was

290.6 milliseconds. Rewriting the create method resulted in an average time of 75.7ms. (a peak time of 66.8ms) which is nearly a factor of four improvement in create performance timing. This ability to modify and enhance the implementation without affecting the abstract specification is a capability entirely unavailable in relational systems, and is clearly a significant advantage for object systems.

Like many of the tests, the range lookup showed a trend of steady improvements in performance with each group of 50, reaching a constant plateau of 54.4 milliseconds after 300 iterations. The average time is comparable to the Rad Unify time.

Performance was not significantly affected by using a remote database. The initial 100 iterations were most affected, as they performed the majority of disk access. Later iterations were unaffected as most of the database had been already cached in memory.

6. Sun Benchmark Criticisms

The Sun Benchmark is useful as a performance metric for comparisons between relational systems, and between relational and object systems. However, it is weak when attempting to provide truly meaningful comparisons between object systems.

The *working set* is atypical for object applications. By a *working set* we mean the set of objects touched or used by the application at any given time during the course of application execution. In this Benchmark each individual benchmark strictly called for objects to be randomly selected and operated upon. Objects could only be clustered with objects of a similar type, but could be kept in the cache over the entire benchmark. In many engineering applications closely related objects are accessed successively, with greater frequency and to a much higher degree than are random, disjoint objects. Because of this general usage pattern, semantically related objects are often physically clustered together in the database. This "semantic clustering" usually results in much higher performance because many of the related objects are brought into cache memory at the time a requested object is brought in, thereby improving the overall access times for related objects.

The Benchmark model (i.e. Person, Author, Document) is quite simplistic and does not attempt to approach the complexity or exercise the usual features that an object-oriented application normally includes. One of the promises of object systems is that they provide features for abstractly modeling complex real world objects and their behavior. To this end, these systems provide things like type hierarchies, inheritance models, complex relationships like A-Part-Of (APO), A-Kind-Of (AKO), An-Instance-Of (AIO), aggregates, inverse properties, versions and alternatives, as well as traditional database features such as concurrency control and multi-user support.

In this simple relational benchmark model there is no need for any dynamic behavior. Modeling dynamic behavior, such as in some kind of event simulation application, is something object systems can perform well and are frequently called upon to model.

These initial criticisms begin to give form to the general criticism that while the benchmark allows database assessment and comparison at a low level of fundamental and common database operations, it does not attempt to address what may be more interesting and meaningful comparisons such as the performance at the level of the application. There may be certain kinds of applications such as engineering design, complex modeling, hypermedia and CASE applications, which are better addressed by object databases. The work by Smith and Zdonik on Intermedia [Smit87] points us in this direction.

To this end, we would like to suggest that future benchmarks allow for examination and assessment of databases at a higher, more complex level. Perhaps they can measure performance more at the level of the application. We would like to suggest that such a benchmark include measurements for the access of large and complex data objects such as documents, and images; that it measure graph or associative traversal operations such closure operations; and that it operate in an environment that more closely approximates an engineering environment with remote databases, and possibly distributed data. Furthermore this application benchmark would need to take into account the kinds of complex modeling relationships object databases have been created to model. Work on just such a benchmark is currently being pursued by [Berre].

7. Closing Remarks

The is paper has offered results for a small version of the Sun Benchmark database. At the time of the writing of this paper, the large version is under construction but is not yet complete.

We have found the Sun Benchmark to be a useful benchmark for measuring simple database operations across different kinds of databases. To some degree, this paper stands as "proof of concept", to the benchmark author's intent of developing a benchmark that can be used for measurements across different kinds of databases. Furthermore, it has allowed us to test and give light to the critical assertion that object databases, such as Vbase, are capable of performing at rates comparable to or faster than existing relational systems.

Lastly, it is evident, that benchmarks that more closely address and assess the capabilities of object databases are needed to properly measure this new and developing technology.

References

[Andr87] Andrews, Tim, Harris, Craig, "Combining Language and Database Advances in an Object Oriented Development Environment", *OOPSLA '87 Conference Proceedings*, SigPlan Notices, Volume 22, Number 12, December 1987.

[Anon85] Anon. et al., "A Measure of Transaction Processing Power", *Datamation*, April 1, 1985.

[Berre] Berre, Arne J. Anderson, T. Lougenia, Porter, Harry, Schneider, Bruce, "The HyperModel Benchmark", Unpublished Work. Oregon Graduate Center, Beaverton Oregon.

[Bitt83] Bitton, D., DeWitt, D. J., Turbfill, C., "Benchmarking Database Systems: A Systematic Approach", *Proceedings of the 1983 VLDB*, 1983.

[Codd70] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 13, 6 (June, 1970) pp 377-387.

[Lisk81] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., Snyder, A., *CLU Reference Manual*, Springer-Verlag 1981.

[Maie86] Maier, David, "Why Object-Oriented Databases Can Succeed Where Others Have Failed", *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, 1986.

[Onto87] Vbase Programmer's Guide, Ontologic, Inc., Billerica MA, 1987.

[Rube87] Rubenstein, W. B., Kubicar, M. S., Cattell, R. G. G., "Benchmarking Simple Database Operations", *SIGMOD '87 Proceedings*, SIGMOD Record, Volume 16, Number 3, December 1987.

[Smit87] Smith, Karen E., Zdonik, Stanley B., Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems", *OOPSLA '87 Conference Proceedings*, SigPlan Notices, Volume 22, Number 12, December 1987.

[Wegn87] Wegner, Peter, "Dimensions of Object-Based Language Design", *OOPSLA '87 Con-*

ference Proceedings, SigPlan Notices, Volume 22, Number 12, December 1987.

Appendix A

Object Version TDL

```
define type Person
  import Document;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    name : String;
    birthdate: Integer;
    publications : distributed Set[Document] inverse
                        Document$authors;
  };

  operations = {
    refines Delete(e:Person)
    triggers (RemoveFromDates);
  };

  define Iterator BirthRange(low : Integer,
                             high : Integer )
     yields (Person)
     method (BirthIter)
  end BirthRange;

  define Procedure Create( T: Type,
    keywords
    name : String,
    birthdate : Integer,
    optional publications : Array [Document],
    optional where: Entity,
    optional hownear: Clustering)
    returns (Person)
    raises (BadCreate)
    method (Person_Create)
    triggers (AddToDates)
  end Create;

end Person;

define Variable Birthdays: btree;

define Type Document
  import Person;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    title : String;
    pages : Integer;
    type  : Integer;
```

```
    date  : Integer;
    publisher : String;
    description : String;
    authors : distributed Set[Person] inverse
                    Person$publications;
  };
end Document;
```

Relational Version TDL

```
define type person
  import Document, Author;
  supertypes = {Entity};
  classtype = $ExplicitClass;
  properties = {
    name : String;
    birthdate: Integer;
        authorlink: distributed Set[Author] inverse
                    Author$personlink;
  };

  operations = {
    refines Delete(e:Person)
    triggers (RemoveFromDates);
  };

  define Iterator BirthRange(low : Integer,
                             high : Integer )
     yields (Person)
     method (BirthIter)
  end BirthRange;


  define Procedure Create( T: Type,
    keywords
    name : String,
    birthdate : Integer,
    optional authorlink : Array [Document],
    optional where: Entity,
    optional hownear: Clustering)
    returns (Person)
    raises (BadCreate)
    method (Person_Create)
    triggers (AddToDates)
  end Create;
end Person;

define Variable Birthdays: btree;

define Type Document
  import Person, Author;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
    title : String;
    pages : Integer;
    type  : Integer;
    date  : Integer;
    publisher : String;
    description : String;
        authorlink: distributed Set[Author] inverse
                    Author$documentlink;
  };
end document;

define type Author
  import Person, Document;
  supertypes = {Entity};
  classtype = $ExplicitClass;

  properties = {
     documentlink : optional Document inverse
                    Document$authorlink;
     personlink : optional Person inverse
                    Person$authorlink;
  };
end author;
```