The VLDB Journal
© Springer-Verlag 1996

An experimental object-based sharing system for networked databases

Doug Fang, Shahram Ghandeharizadeh, Dennis McLeod

Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781, USA

Edited by Georges Gardarin. Received October 29, 1992 / Revised May 4, 1994 / Accepted March 1, 1995

Abstract. An approach and mechanism for the transparent sharing of objects in an environment of interconnected (networked), autonomous database systems is presented. An experimental prototype system has been designed and implemented, and an analysis of its performance conducted. Previous approaches to sharing in this environment typically rely on the use of a global, integrated conceptual database schema; users and applications must pose queries at this new level of abstraction to access remote information. By contrast, our approach provides a mechanism that allows users to import remote objects directly into their local database transparently; access to remote objects is virtually the same as access to local objects. The experimental prototype system that has been designed and implemented is based on the Iris and Omega object-based database management systems; this system supports the sharing of data and meta-data objects (information units) as well as units of behavior. The results of experiments conducted to evaluate the performance of our mechanism demonstrate the feasibility of database transparent object sharing in a federated environment, and provide insight into the performance overhead and tradeoffs involved.

Key words: Database system interoperability – Object sharing – Experimental prototype benchmarking

1 Introduction

Data and knowledge base system interoperability is increasingly important, as computer systems and their associated collections of information proliferate, and as global connectivity becomes more and more a reality. We address this essential area of research by considering an environment consisting of a network of data/knowledge bases and their supporting systems, and in which it is desired to accommodate the controlled sharing and exchange of information among the collection. Such networked/federated database environments are common in various application domains, including office information systems, next generation libraries, computer-integrated manufacturing systems (with computer-aided design as a subset), personal computing, and scientific research information bases. New approaches and techniques to support the interoperation of such systems are required, while at the same time respecting the autonomy of the individual component database systems. In order to have significant practical impact, these approaches must be transparent, flexible, efficient and place minimal requirements on (existing) component database systems. We take an object-based approach to sharing in this networked database context. In addition to sharing data objects (information units) at various levels of abstraction, our approach also supports the sharing of units of behavior (operations, methods, or functions).

The foundations for our work arise from research on heterogeneous databases (Shan 1989; Smith 1981; Stonebraker 1977; Templeton 1987), federated databases (Heimbigner 1985; Sheth 1990; Gardarin 1995) and multidatabases (Litwin 1986). The heterogeneous database approach typically relies on a single monolithic global schema. Users are required to participate at this new level of abstraction (or a derived view of it) in order to access shared data. Some limitations of this approach are apparent: users access shared data with new tools (i.e., there is a lack of database transparency), and limited flexibility is provided, since the requirement for a global schema mandates a single representation of data. The federated approach stresses autonomy and flexible sharing patterns through inter-component negotiation. Rather than utilizing a single, static global schema, the federated architecture allows multiple "import" and "export" schemas for component database systems. The multidatabase architecture is in a sense similar to the federated architecture. Emphasis is placed on the interoperability among component databases based on a flexible, common multidatabase language. In this approach, the user is responsible for keeping track of the various databases and their schemas in order to navigate and manipulate data. In addition, the user utilizes a new multidatabase language to manipulate the shared data.

By contrast, our approach allows each component to integrate objects directly into its local database, thereby maximizing flexibility and providing database transparency. Furthermore, our object-based approach allows for both fine and coarse grain sharing, as well as the sharing of behavior.

With particular regard to the sharing of behavior, the approach and mechanisms described here are consistent with the approaches taken in CORBA (Object Management Group 1991; OMG 1992), but are much more flexible. The type of behavior sharing supported in CORBA is based upon the encapsulation of operations at an object's interface; these operations are registered in an "object request broker" implementation using the OMG interface definition language. Clients can request a service from an object by specifying the operation, an object reference, and any additional necessary parameters. The requested service is actually executed remotely in an object implementation in which the object resides, and the result is sent back to the client. Our approach supports a more flexible paradigm, in which object and the operations that manipulate them can both be either local or remote.

In what follows, we describe a mechanism and experimental system to support the various kinds of sharing patterns that arise in the context of an object-based sharing model. We specifically describe and analyze a multi-configurational experimental prototype system that we have constructed to demonstrate, refine, and evaluate the techniques devised.

The research described in this paper is couched in the context of a larger effort, which is addressing three key aspects of sharing and interconnection in networked database systems. These may be viewed without loss of generality in the context of a given component (C), which intends to import information from other (remote) components: (1) the discovery and identification by component C of relevant non-local information (Hammer 1994); (2) the resolution of the similarities and differences between C's information and relevant non-local information (Hammer 1993); and (3) the efficient realization and implementation of actual sharing and transmission of information to and from C and other components. The focus of this paper is on the third of these key issues.

It is important to note that the research described in this paper focuses on the underlying techniques and mechanisms to allow database users to import and export objects. We are not, for example, specifically concerned with providing complete facilities for remote update or global consistency; these are large research problems in their own right and are being actively investigated by other researchers. We also do not directly address here the problem of determining whether objects in two databases refer to the same real-world entity (this problem is considered to be at a higher level than the focus of this paper) (Kent 1993). Finally, it is of course well recognized that many social and legal issues remain open; policies regarding the fair use of information, copy and ownership rights, and security need to be addressed for different environments (scientific versus commercial). This paper strives to provide a collection of mechanisms that enable the users to share and exchange information transparently in the presence of multiple repositories. The implementation of a final system (along with its optimization and performance specs) depends on the target environment and its specific constraints.

The remainder of this paper is organized as follows. Section 2 briefly presents the functional object-based model that serves as an inter-component sharing forum. In Sect. 3, we provide a unified framework for inter-component sharing, based upon our functional approach. Section 4 describes the sharing mechanism, specifically supporting instance, type, and behavior sharing. In Sect. 5, we examine in detail the experimental prototype system we have built. Section 6 provides a substantive analysis of the performance of our experimental system, and analyzes the general impact of our observations based upon our experimental results. Finally, Sect. 7 presents concluding remarks and a brief discussion of research directions.

2 An object-based context for sharing

A generic functional object-based data(base) model is employed here as a basis for inter-component sharing and information unit exchange. This model supports the usual object-oriented constructs. In addition, the constructs provided in this data model serve as a reference point from which we can later describe our techniques for transparent sharing. We also describe three sharing patterns (instance, type, and behavior) that naturally arise in the object database context and illustrate them using this data model.

2.1 A functional object-based model

The conceptual database model considered in this work draws upon the essentials of functional database models, such as those proposed in Daplex (Shipman 1981), Iris (Fishman 1987), and Omega (Ghandeharizadeh 1993b). Our functionally object-based model contains features common to most semantic (Afsarmanesh 1989; Hull 1987) and object-oriented database models (Atkinson 1989), such as GemStone (Maier 1986), O_2 (Lecluse 1988), and Orion (Kim 1987). In particular, the model supports complex objects (aggregation), type membership (classification), subtype to supertype relationships (generalization), inheritance of functions (attributes) from supertype to subtypes, run-time binding of functions (method override), and user-definable functions (methods).

In this model, functions are used to represent inter-object relationships (attributes), queries (derived data), and operations (methods). Two types of functions can thus be distinguished:

1. *Stored functions*: A stored function records data as primitive facts in the database. Stored functions can be updated.
2. *Computed functions*: A computed function (sometimes termed a foreign function) is defined by a procedure written in some programming language. The value of a computed function cannot be directly updated.

To illustrate our diagrammatic notation, consider the example of two collaborating researchers, Researcher-A and Researcher-B. Each researcher maintains separate databases of journal and conference publications using a different underlying schema to model this information. Figure 1 represents the conceptual schemas of Researcher-A's and Researcher-B's databases. In our diagrammatic notation, instance and type objects are depicted as bubbles. For type

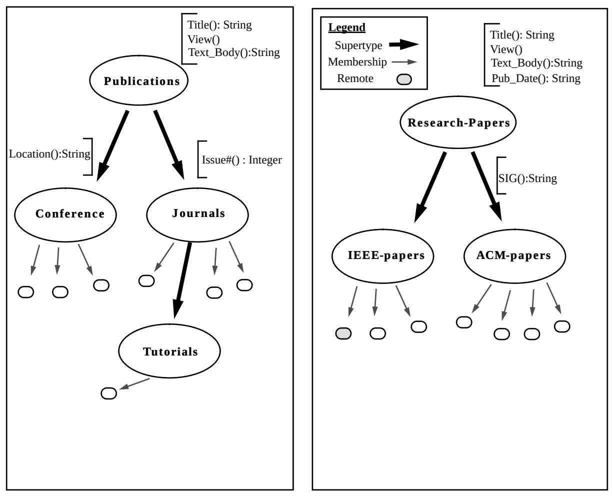

Fig. 1a,b. Two-example component database schemas. **a** Researcher-A's conceptual schema. **b** Researcher-B's conceptual schema

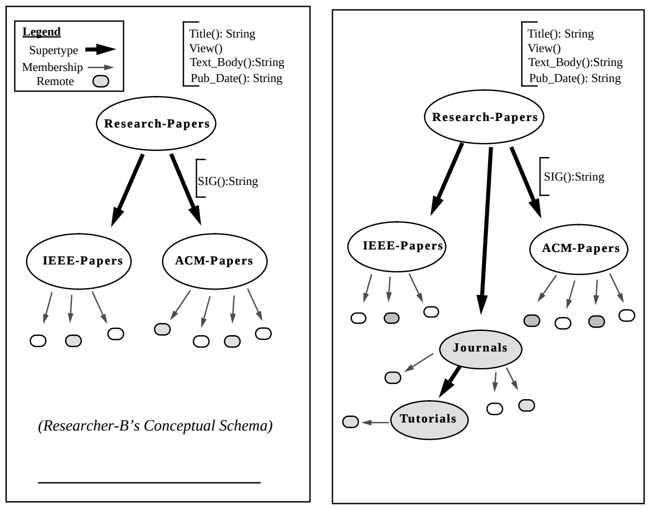

Fig. 2a,b. Sharing pattern example. **a** Sharing instance objects. **b** Sharing type objects

objects, the name of the type is placed within the bubble. Each type has a set of functions defined on it. The signature of each of these functions is placed immediately above the type. The input arguments of each of these functions are instance objects of the type upon which the function is defined. Hence, the input argument type of functions is not shown in their signatures; it is assumed to be of the type on which the function is defined.[1] Hence, in Fig. 1a, the functions defined on the type **Publications** are: *Title()*, *View()*, and *Text_Body()*.

Two kinds of inter-object relationships are explicitly modeled and have corresponding diagrammatic notations: the supertype to subtype (with inheritance) interclass relationship and the type membership relationship. All other inter-object relationships are modeled through functions. The supertype to subtype relationship is depicted with thick dark lines from supertype object to subtype object. In Fig. 1a, **Conferences** and **Journals** are subtypes of the type **Publications**. Type membership is depicted with thin dotted lines from type objects to its members (i.e., instances). In Fig. 1a, the **Journals** type has four instances: three directly created as **Journals** and one from the **Tutorials** subtype.

2.2 Object sharing

Since every object is treated uniformly in our data model, it is natural to investigate the sharing of individual data objects (instances), structural objects (e.g., types), and behavioral objects (functions). Recall the example of collaborating researchers; suppose that Researcher-B would like to import some specific publications from Researcher-A. This situation corresponds to instance sharing and is illustrated in Fig. 2a. Imported instance objects are denoted by the hashed bubbles in the figure. Hence, Researcher-B now sees four instances of **ACM-papers** and three instances of **IEEE-papers**, where originally there were only two instances in each type. In our approach, all the instances appear local (i.e., transparent) to Researcher-B, even though Fig. 2a depicts the remote instances differently than local ones.

Existing approaches have concentrated on what corresponds closely to type sharing (Stonebraker 1988; Williams 1981). These systems focus on the sharing of entire collections (e.g., relations), rather than members of the collection (e.g., tuples). By contrast, importing a remote type naturally leads to importing the subtree of the remote type hierarchy rooted at that type. Again, using the collaborating researchers example, assume Researcher-B would also like to have access to all of Researcher-A's journal publications. Figure 2b illustrates this situation. In this case, Researcher-B also imports the **Tutorials** type by virtue of the supertype-subtype relationship. The principal advantage of this kind of sharing is that it allows Researcher-B to use Researcher-A's database without integrating Researcher-A's conceptual schema with his/her own conceptual schema; Researcher-B simply uses the part of Researcher-A's original schema rooted at **Journals**. Another way of looking at this is that Researcher B can now move from his/her own context to Researcher A's context. Researcher B can even add or delete instances from his/her own context of Researcher A (note that one of the instances of **Journals** in Fig. 2b is local) without updating Researcher A's database. This is a useful feature when the autonomy constraints of Researcher A prohibit any addition or deletion of its instances. To some degree, instance sharing can be viewed as the complementary situation; instead of integrating local instances into a remote component context, the goal is to integrate remote instances into the local component context.

In addition to simply sharing objects representing information units, it is also possible to share behavioral objects. This enables the importer to access services not provided by his/her local system. For example, in the collaborating researchers scenario, assume Researcher-B has a dvi previewer method but no LaTeX compiler on his/her local system. Further, Researcher-A has LaTeX available by virtue of a function, *latex()*, which takes ASCII text with LaTeX

[1] The function can consume the instances of its subtypes using inheritance

commands as input and outputs dvi format. In this case, our approach allows Researcher-B to share the function *latex()*. Sharing functions in this way is very natural in our data model because functions are represented as objects.

3 A unified framework based on function sharing

To provide a top level, unifying view of sharing in the object database context, we consider a functional viewpoint on sharing. From this point of view, an object is an entity with its own identity; an object unites the values of its different properties. Functions are applied to an object in order to access values associated with that object (i.e., its state). These functions can be *stored* (i.e., attributes) or they can be *computed* (i.e., methods). Using this perspective of the relationship between objects and functions, we can consider the effects of distribution on both objects and the functions (stored or computed) independently.[2]

3.1 A Taxonomy of Function Sharing

Let us assume the existence of a function $\mathscr{F}$, which can be shared among components of a federation; without loss of generality, assume $\mathscr{F}$ takes as input the argument a.[3] The argument type can be a literal (i.e., **Integer**, **String**) or a user-defined type such as **Research-Papers**. Sharing takes place on a component-pairwise basis, meaning that $\mathscr{F}$ is exported by a component $C1$ and imported by a component $C2$. The importing component is called the *local database*, while the exporting component is called the *remote database*. There are several ways in which components $C1$ and $C2$ can share the service provided by $\mathscr{F}$, depending upon the location where $\mathscr{F}$ executes and upon where its input argument a resides (i.e., there are two degrees of freedom). Hence, at this level of abstraction there are four distinct *function-argument* combinations depending on whether $\mathscr{F}$ or a is local or remote. In addition, since functions can be further differentiated as either stored or computed functions, we can now distinguish between a total of eight different sharing scenarios. Some of these scenarios are trivial, but we present them below for the sake of completeness. We first focus on stored functions and then turn our attention to computed functions.

3.1.1 Stored functions

As a framework for analysis, recall the collaborating researchers scenario where both Researcher-A and Researcher-B maintain separate databases of journal and conference publications. Figure 1 shows both schemas together. In this scenario, Researcher-B is the local database and Researcher-A is the remote database. The four situations for the sharing of stored functions among components can be broken down as follows.

[2] See Fang (1992a) for more details

[3] Since the argument can be a complex unit of information, this is not a limitation; multiple arguments can be handled by an obvious extension of our approach

Local function – Local object. This is what we term the “base case”. Both objects, the stored function $\mathscr{F}$ and its argument a, reside in the local component and can be executed as usual; all processing of $\mathscr{F}$ is done locally.

Local function – Remote object. In this case, the local stored function $\mathscr{F}$ is applied to argument a which resides remotely. This situation has the effect of giving local state to a remote object. For example, Researcher-B can create a value for the Pub_Date() function on the highlighted remote object in Fig. 1. The value of Pub_Date() is stored locally in Researcher-B’s database while Researcher-A remains unaware of the existence of Pub_Date(). This feature is very useful for allowing Researcher-B to “adjust” remote instances and customize them to the local environment while respecting the autonomy of the Researcher-A’s database.

Remote function – Local object. This situation is somewhat meaningless, since stored functions only have a meaning in the local context of the component in which they were initially created.

Remote function – Remote object. Similar to the first case, this is also a base case; both the state of the object and the execution of the function are in the same component (e.g., Researcher-A). The difficulty here lies in providing database transparency, as discussed immediately below. We point out here that this situation forms the basis of the instance sharing pattern described in Sect. 2.2. For example, the shaded remote instance of Fig. 1 would appear local and have its original values for the *Title()* and *Text_Body()* functions that are defined in Researcher-A’s database.

3.1.2 Computed functions

As in the case for stored functions, there are four combinations of where the computed function executes and where the object resides. Below we analyze each case more closely.

Local function – Local object. As in the case of stored functions, this is the base case. Computed function $\mathscr{F}$ as well as its argument (a) reside in the local component, and the execution is local.

Local function – Remote object. This situation can be reduced to the base case described in case 1. For example, Researcher-B can use his/her own locally defined *View()* computed function to view the remote instance in Fig. 1.

Remote function – Local object. This is the reverse of the previous case: the function executes remotely and the input argument is supplied from the local database. In effect, the remote database is providing a non-local “service”. Intuitively, this is the most useful scenario from Researcher-B’s perspective and forms the basis for behavior sharing. For example, if Researcher-B did not have a *View()* function, then s/he could use the *View()* defined in Researcher-A’s database.

Remote function – Remote object. This situation is similar to the first case (Local function – Local object) in that both the state of the object and execution of the function are in the same component. For example, Researcher-B views one of

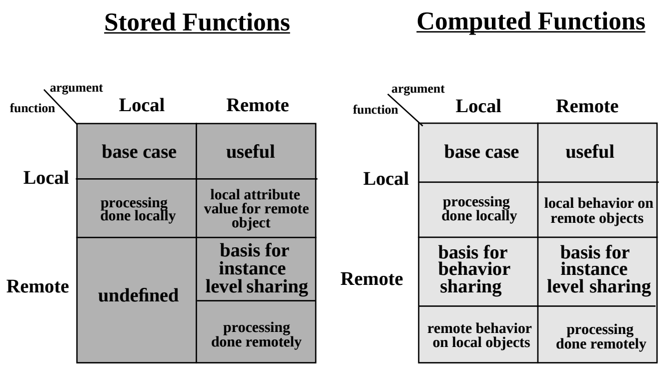

Fig. 3. Function sharing taxonomy

Researcher-A's **Conference-Papers** using Researcher-A's original *View()* function.

In the examples above, the functions have returned a literal type [e.g., the *Title()* function returns a **String**]. However, functions with signatures involving abstract (user-defined) types can also be shared. In this case, both the input and output argument types must be defined locally; if they are not, their meta-data must be imported beforehand. The location of the result argument is determined by the location where the function executes.

3.2 The taxonomy and object sharing

Given the taxonomy presented above, we can now consider how it can be used to unify the original notions of instance, type, and behavior sharing introduced in Sect. 2.2. We proceed by explaining how each of these sharing patterns can be implemented using a case presented in the taxonomy. The two tables in Fig. 3 summarize these results.

Conceptually, in instance sharing, a remote instance object is imported directly into a local type. This remote instance behaves in the same manner as a local instance object from the user's perspective. However, the actual state of the remote instance exists in the remote component database; retrieval of any state of the remote object is accomplished by accessing the remote database transparently. Hence, access to remote instance objects corresponds to the *Remote function – Remote object* situations described in Fig. 3.[4] Sharing behavioral objects corresponds to sharing a computed function that exists on a remote component. Intuitively, when an instance object is imported, only data is being shared. On the other hand, importing a behavioral object gives the importer access to services not provided by his/her local system. This corresponds to the *Remote function – Local object* situation in the taxonomy for computed functions. Type sharing consists of two aspects: (1) sharing the meta-data, and (2) sharing all the instances of a type. The first aspect does not relate to our taxonomy. The second aspect is handled by instance sharing.

One particularly useful sharing pattern revealed by the taxonomy, but not directly covered by instance or behavior sharing is the important case of *Local function – Remote object*. Among other things this situation allows users to add additional state to remote objects without modification of the exporting database, thereby preserving the autonomy of the exporter. This ability to create local state for remote objects is achieved automatically from the way we support instance sharing, and corresponds to simple local database access. It also allows a remote object to be customized to the environment of the local component database.

[4] Note that this applies to both stored or computed functions

3.3 Discussion

In more practical terms, our taxonomy of eight different sharing patterns can be reduced to two "most interesting" cases: (1) executing an imported function on a local argument; this corresponds the *Remote function – Local object* situation; and (2) executing a local function on an imported (shared) argument; this corresponds to the *Local function – Remote object* situation. The first case only applies to computed functions and can be described as reusing a previously defined function from another component in the federation; software reuse is the primary reason for components sharing behavior. The second case applies to both computed and stored functions and can be described as extending the "characteristics" of a remote object with added functionality while at the same time respecting the autonomy of the originating site. In this case, the structure of an object (type) is shared by other components which will not be able to see or modify the original object.

4 A sharing mechanism

Given the function-based framework for sharing described above, we now present a mechanism for implementing object sharing (instance, type, and behavior). In our discussion of function sharing, we have stressed the separation of the location where the function executes from the location where the data resides. However, in order to achieve database transparency, this separation of function execution and argument location should be completely transparent to the user. A major additional goal of our sharing mechanism and its implementation is therefore to achieve database transparency for the instance, type, and behavior sharing patterns.

4.1 Instance sharing

Our mechanism for instance sharing relies on creating surrogate objects in a database for each remote object that it imports. Surrogates are simply objects that are created locally and serve as place holders for remote objects. Thus, the local database management system is able to interpret and manipulate them as any other local object. Using these "surrogates" alone, however, is not enough. The functions encapsulating surrogate objects must be overridden to use computed functions which access the remote component where the object is actually stored. In order for the overloading to be performed correctly, function naming, binding, and placement in the type hierarchy are critical (Fang 1992b).

To illustrate, assume that Researcher-A has made his/her publications available for other users, and that Researcher-B has imported this data. Although the two schemas are

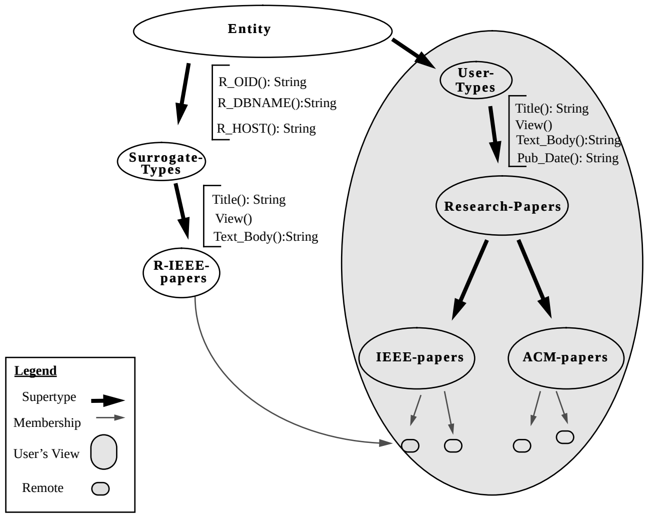

Fig. 4. Sharing instance objects

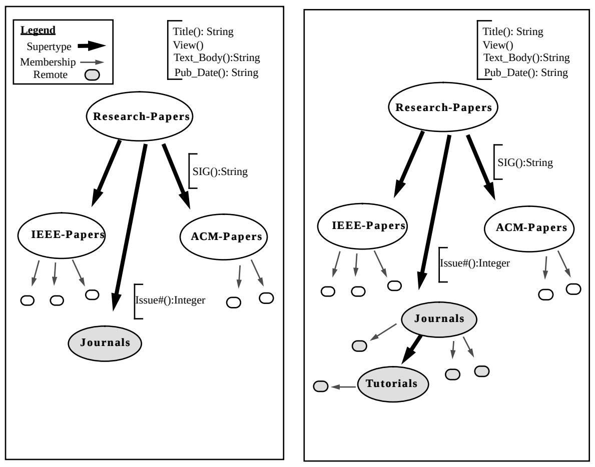

Fig. 5a,b. Sharing type objects. **a** Import meta-data of journal. **b** Import journal type

different, each **Publication** instance object in Researcher-B's database has a similar notion of *Title()*, *View()*, and *Text_Body()* as Researcher-A's instances. Hence, only those functions need to be imported into Researcher-A's database. Figure 4 shows one instance object of Researcher-A being imported into Researcher-B's database. The large gray bubble in the diagram indicates Researcher-A's perspective of the conceptual schema. Researcher-B is unaware of the other types in the diagram. Our technique for importing remote instance objects can be summarized in the following three steps:

1. Create a local object (i.e., surrogate) for each remote instance object.
2. Create computed functions that access and retrieve data from remote components.
3. Force functions defined on surrogates to use (refer to) the newly created computed functions in step 2.

A surrogate serves as a local handle to access a remote object. By using the surrogate, differences between remote representations of objects, e.g., object identifiers (OIDs), can be masked out (made transparent). Since the state of the remote object exists externally, computed functions that make remote procedure call (RPC) (Birrell 1984) requests to access that state are created. Finally, in order to have the surrogates use the remote functions, any existing functions defined on the surrogate must be overridden to use the RPC defined functions. This can be implemented by dynamically binding functions to objects.

Surrogates are created as instances of both a local type and a surrogate type. The purpose of the surrogate type (e.g., **R-IEEE-Papers**) is to override the local functions that surrogates inherit from the local type of which they are a member. By additionally creating the surrogate as a member of this surrogate type, the functions that the surrogate instance originally inherited are overridden. The two thin dotted arrows from **R-IEEE-Papers** ("R" for remote) and **IEEE-Papers** to the surrogate instance serve to indicate that surrogate instances (i.e., remote instances) of **IEEE-Papers** are created as members of both **IEEE-Papers** and **R-IEEE-Papers**. Thus these (remote) instances inherit multiple functions from both **R-IEEE-Papers** and **IEEE-Papers**; any duplicately named function from the two types is overridden by the function defined for **R-IEEE-Papers**. The functions defined on **R-IEEE-Papers** are computed functions, which make RPC requests to the remote component database and retrieve the values of functions on remote instances. This approach exploits the fact that objects can be members of more than one type (e.g., IEEE-Papers and R-IEEE-Papers). Dynamic binding is used to resolve overloaded functions so that the functions defined by **IEEE-Papers** on a surrogate are actually overridden to be those defined on **R-IEEE-Papers**.

An alternative approach to the above would be to create **R-IEEE** as a subtype of **IEEE-Papers** and create the surrogates as instances of **R-IEEE**. One major drawback of this approach is that remote instances are no longer transparent members of the **IEEE-Papers**. The user is now made aware of the **R-IEEE** subtype in his/her conceptual schema. A second drawback is that in this approach surrogate types appear scattered throughout the user's conceptual schema. In our original approach, all surrogate types are contained to a single branch of the type hierarchy, separate from the user's conceptual schema.

4.2 Type sharing

There are two aspects to sharing type objects. The first is to share only the meta-data associated with the type being imported. The second aspect is to also include all the members and subtypes of the type being imported.

The meta-data associated with a type consist of the signatures of the functions that serve to encapsulate the members of that type. Hence, all that is needed to import the meta-data of a remote type is to create a surrogate type with the same functions. The only real complication occurs when one of the functions has a signature whose result type is not known locally. Conceptually, it is not difficult to apply this simple rule recursively to create a surrogate type for the closure of this type. However, the entire closure may not necessarily be required. The results of importing only the meta-data is illustrated in Fig. 5b for the type **Journal**.

In addition to importing meta-data, the second aspect of type object importation involves also importing the remote members and subtypes of the type. This is the kind of type sharing illustrated in Fig. 2b. To accomplish this, we essentially use the same paradigm as described earlier: (1) import meta-data of a type, and (2) import all the instances of the meta-data as in instance sharing. To include subtypes, this rule can be applied recursively. Figure 5b illustrates this kind of type sharing for the type **Journal**.

4.3 Behavior sharing

The goal of supporting the sharing of behavioral objects is to allow a component to utilize remote services that may not be available locally. This corresponds to sharing *computed* functions.[5]

As in instance sharing, meta-information containing the location (e.g., remote OID and remote component name) of the remote function object being imported must be stored locally. However, in contrast with the case for instance sharing, this meta-information is associated directly with the function being imported. In instance sharing, the meta-information is indirectly kept for remote functions such as *Title()* and *View()* via the surrogate instance object (see Fig. 4). Thus, in our implementation we distinguish between two kinds of "remote" functions: those implicitly defined by importing instance objects and those directly imported when behavioral objects are shared. Figure 6 shows our mechanism for incorporating this meta-information for sharing behavioral objects. We exploit the fact that meta-data is also represented using our functional object-based model. Imported functions are created as instances of the type **Remote-Functions** and can thus store and access the additional location meta-information required to execute the imported function.

We now present a slightly different example from Fig. 4. In this scenario (Fig. 6), the *View()* function is imported from some remote component. In addition, **R-IEEE-Papers** no longer supplies a *View()* function. Hence both the remote and local instances of **IEEE-Papers** use the same imported *View()* function for displaying research papers. This is evident in Fig. 6 by the absence of the *View()* function from the type **R-IEEE-Papers** and the addition of a new (italics faced) *View()* function defined on **Research-Papers**.[6]

In order to explain how the *View()* function works, we must first explain how our implementation addresses the issue of side-effects. By side-effect we mean two things:[7] (1) any kind of implicit input other than the input argument that is necessary to compute the result, and (2) any modifications to the state of the database where the function executes other than to the input argument. For functions whose arguments

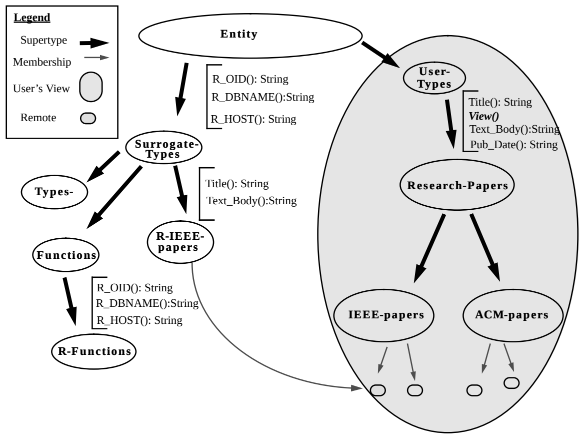

Fig. 6. Sharing function objects

are literals, this simply requires that the function being imported computes its result value solely based on its input argument without modifying any database state [e.g., *Fibonacci()*]. Functions whose input argument are non-literals pose additional difficulties. In this case, the input argument is the OID of an instance. The problem then lies in determining what information a computed function accesses in order to compute its results. Strictly applying our definition of side-effects would restrict computed functions on non-literals to solely accessing and then manipulating the input OID. But realistically, a computed function must be able to access some state of the object corresponding to the OID when computing its result. In our implementation, we take the position that the only state that a computed function can access are those functions that serve to encapsulate that object. In other words, the only state the computed function will possibly access are those functions that are defined on the types of which the instance is a member. In the case of the *View()* computed function, the only functions that *View()* needs to access are *Title()*, *Text_Body()*, and *Pub_Date()*.

Having determined what information a computed function on a non-literal type can access, a problem arises when trying to execute such a function remotely on a local object. The problem occurs when supplying local arguments to a remote computed function. Although we know the computed function is limited to only accessing the functions that encapsulate the instance, we do not know exactly which ones it does need. Even if we pass all the possible values the computed function can access, the computed function must be written in such a way as to retrieve these arguments from the network and not the local database. This would be undesirable and contrary to our goal of relieving the computed function writer of needing to know where the data on which it operates is located.

The best approach to this problem in our autonomous environment is to allow computed function writers to define functions without concern as to whether the function is to be exported. Achieving this tighter level of coupling between components requires additional functionality on the exporter. We briefly discuss below a simple "callback" mechanism implemented in the Omega prototype that will handle computed functions that access the object's own stored functions. Whenever a remotely executing computed function needs

[5] This corresponds to the computed function case of *Remote function – Local object* in the taxonomy presented in Sect. 3

[6] The italics font is used to indicate that the *View()* function is imported and no longer local

[7] This extends the more traditional definition given by the programming language community which defines side-effect as "the modification of a data object that is bound to a non-local variable"

state (i.e., a stored function) from the local database, it performs a callback to the local database to retrieve that state. In particular, the local database that imports remote objects can be viewed essentially as a client and the exporter providing the objects as a server. During instance sharing, the local database simply operates as a client and makes RPC requests to the server. However, for behavior sharing, when the callback mechanism is used, the local database must in addition operate as a server to accept the callback requests. Computed functions can be written using any programming language that can be compiled to re-entrant object code. This object code is then dynamically linked into the database management system kernel when the computed function is accessed. In our prototype using the Omega database management system, a computed function accesses the local database through *Omega_eval()*, which has two parameters: an argument and the function that is to be applied to the argument.[8]

We can now consider how the callback mechanism works transparently and allows computed functions to be written uniformly without regard as to whether that function is to be exported. Consider again the example using the *View()* computed function. Suppose that the imported *View()* function retrieves the *Text_Body()* of an instance (say in LATEX format), computes the dvi formatted version, and displays the formatted version through a dvi previewer (e.g., xdvi). When the user of the local database invokes the *View()* function on a local object, the *View()* function is passed the local OID of a **Research-Papers** instance. The *View()* on the remote server makes a call to Omega_eval() to retrieve the *Text_Body()*, Omega_eval() recognizes that the OID passed in as its argument is not local and performs a callback to the server of the local database that invoked the *View()* function. Since the local server recognizes the OID as a local OID, it performs the request and passes back the *Text_Body()* to the remote server which can than complete its computation and display the results on the local database monitor.

Functions involving signatures returning complex/abstract data types can theoretically also be imported. In this case the result argument type must be defined locally; if they are not, their meta-data can be imported using *Import_Meta()*.

5 Experimental prototype implementation

An experimental testbed for our sharing mechanism has been designed and implemented, supporting interconnected components based on the Iris and Omega object-based database management systems. In our testbed, a component consists of a database and is associated with a machine where the database physically resides.[9] A component can participate in as either the importer or exporter of information. In the first case, objects from other components are seamlessly integrated with the local objects in the importer's database. When a component acts as the exporter of information, other components can access and import its information (e.g., data or behavioral objects). In what follows, we discuss the implementation details of both exporter and importer functionalities in our Iris and Omega prototypes.

5.1 The Iris prototype

For our Iris prototype, we used an early version of the Iris database management system that was developed at Hewlett Packard Laboratories (Fishman 1987). Iris is based on a functional object-oriented data model which supports the usual object-oriented constructs. The data manipulation language provided with Iris is a variant of SQL called OSQL. In the following, we discuss key stages involved in implementing the sharing mechanism described in the previous section for an Iris based component.

In order to implement our sharing mechanism, we extended Iris in three ways. First, our approach relies on overriding stored functions with computed functions. Writing a separate computed function for each overloaded stored function would have been tedious and inefficient. In addition, the overhead in dynamically loading each function would have been significant. We eventually wrote a single foreign function which applies a remote function to a remote OID, given the host and database names upon which the function and instance exist. This function, *r_ieval()* (remote iris-eval), was written using the SUN RPC protocol (Williams 1981). All other computed functions for accessing remote data were derived from it. Second, instead of using dynamic binding, Iris by default uses compile-time (early) binding of types to the variables being quantified over in the *for each* clause of an OSQL query for resolving overloaded functions. To illustrate how this affects our mechanism, consider the following query posed against the collaborating researcher's example database depicted in Fig. 4.

```
select Title(r-paper)
for each Research-Papers r-paper;
```

With early binding, the variable *r-paper* is statically bound to the type **Research-Papers**; thus, the same *Title()* function (the one defined on **Research-Papers**) would be applied to every instance **Research-Papers** including the instances of its subtypes, **IEEE-Papers** and **ACM-Papers**. However, for remote instances (e.g., the shaded instance of **IEEE-Papers**) in Fig. 4, our mechanism requires the overloaded *Title()* function to be resolved to using the one defined on **R-IEEE-papers**. Fortunately, since Iris is an extensible system, we can override this functionality; indeed, Iris provides a function *latebind* which uses the needed semantics. Finally, our mechanism requires a server which would service RPC requests when Iris is used as the exporter.[10] Instead of creating one session for each request, the server opens a single session and services all requests made by an importer within that session (each request is treated as a transaction). After a certain period of non-use, the server times out, closes the session and returns to the initial state. The only functionality required from the server is the ability to: (1) receive a RPC

[8] Omega_eval() is described in more detail in Sect. 5.2

[9] For the sake of simplicity, we will assume that each component exports or imports only a single database and that each component resides on a single machine

[10] Rpcgen (Sun Microsystems 1988) was utilized to simplify the construction of both the client stub and server and to facilitate access to other RPC services

request, (2) apply a function to an argument, and (3) return the results to the requester. The resulting simplicity of this interface greatly facilitates the addition of components to the federation.

5.2 The omega prototype

Omega (Ghandeharizadeh 1993b) is a parallel object-based system constructed using a relational file system [the Wisconsin Storage Structure (Chou 1985)]. Similar to Iris, it is based on a functional data model (Shipman 1981) and supports OSQL at its data manipulation language. We used a single-node configuration of Omega for the testbed prototype. This configuration allows evaluation and testing of the prototype to be much simpler. In the following, we present the implementation of Omega[11] as it relates to some of the key constructs needed in implementing our mechanism. In addition, this overview will prove useful when we interpret the results of the experiments performed in the next section.

The physical implementation of Omega is organized around three types of files: (1) a *meta-data* file, (2) a *Sub-object Directory (SubDir)* file, and (3) a *t-file* for each type in the system. Omega represents a type as a record in the meta-data file. This record contains: the name of a type, its immediate subtypes (supertypes), the identity of processors that contain its fragments, relevant declustering information, number of functions, name of each of its functions, internal organization of a function, available indices on a function, etc.

Omega represents an object as a collection of sub-objects. A sub-object represents the membership of an object in a type. The system supports a t-file for each type in the system and groups its instances (each instance is a sub-object of an object) together in that file. An object is represented as a record in the SubDir file. This record maintains a set of (t-file,SID) pairs where a t-file corresponds to a type and SID is a physical pointer to the sub-object that represents the membership of this object in that type. The physical location of this record constitutes its identifier that uniquely distinguishes this object from the other objects in the system (OID).

The organization of the SubDir file allows Omega to support objects that are members of more than one class. In addition, the SubDir file provides an efficient way for implementing dynamic binding. In order to support dynamic binding semantics, the exact order in which an object becomes a member of a type must be maintained. This is done by ordering the set of (t-file,SID) pairs for an object from left to right. Each time a new type is added to the object, the new (t-file,SID) pair is simply appended to the rightmost position of the object's record in the SubDir file.

Omega supports *stored* and *computed* functions. Evaluating the value of a function for a particular argument depends on its implementation. Retrieving the value of a stored function requires a record look up in the corresponding t-file. Obtaining the value of a computed function on the other hand, requires first dynamically loading the program into memory and then executing the program. In order to abstract out the implementation of a function, the all-purpose function, *Omega_eval()*, was created. This function takes in as arguments the name of a function and the function's argument. Next, it applies the function to the argument and returns the results. Dynamic binding is implemented in Omega_eval() by scanning the SubDir record for an object from right to left. The first type which has the required function is the one that is applied to the object.

The Omega importer was constructed exactly as described in Sect. 4 using the same RPC stub generated for the Iris component. Similarly, the Omega exporter was constructed using a server similar to the Iris prototype that makes direct calls to Omega_eval().

[11] For a more detailed description see Ghandeharizadeh (1993b)

6 Experimental analysis

A substantive experimental evaluation of the performance of our Iris and Omega has been completed. The goal of this evaluation was to quantify the overhead of our sharing mechanism, and to identify the central issues and tradeoffs involved in object-based sharing in general. In overview, we observed a wide variation in the relative overhead of our mechanism between the Iris and Omega prototypes. In order to study these factors more systematically, we instrumented the Omega prototypes and conducted a second series of experiments using only Omega components.[12] For these experiments we chose to use the USC Benchmark (Ghandeharizadeh 1993a), a benchmark tailored for object-based database systems because it was readily available. [Other benchmarks would also have been well suited to our needs, e.g., (Anderson 1990; Cattell 1988).

In the following, we provide a brief description of the USC synthetic database. Next, we present the results of our experiments for two types of queries: (1) those that process the instances of a type, and (2) those that reference a fixed set of functions. For the first type of queries, we controlled the percentage of the instances that are imported from a remote site (horizontal partitioning), while for the second type of queries we controlled the number of functions imported from a remote site (vertical partitioning). We conclude by analyzing these results and presenting our main conclusions about the overhead due to our mechanism and the factors affecting the overall performance of the system.

6.1 The benchmark database

We used a modified version of the USC benchmark for this experimental study. The database consisted of a single type, LV0-T-ROOT containing MAXOBJS instances. Table 1 contains the functions defined on LV0-T-ROOT. The first nine functions can be used to model queries with a wide range of selectivity factors. The name of each function reflects its range of values. UID(), for example, is an integer ranging between [0 - MAXOBJS-1] and is assigned sequentially during object creation time. The last three functions are object-valued and reference an object of the same type. *One-to-one()* is a single valued function that maps one object to

[12] We could not instrument the Iris prototype because we did not have access to its source code

Table 1. Functions defined on the root of the usc benchmark type lattice

Function Name	Return Type	Range of Values	Order
Unique Identifier (UID)	Integer	0-(MAXOBJS - 1)	Sequential
Shuffled UID (SH-UID)	Integer	0-(MAXOBJS - 1)	Random
Unique Float (UF)	Float	0-(MAXOBJS - 1)	Random
Unique String1 (USTR1)	String (52 bytes)	'**...*UID'	Sequential
Shuffled String2 (SH-USTR2)	String (52 bytes)	'**...*SH-UID'	Random
one-percent	Integer	0-99	Random
ten-percent	Integer	0-9	Random
even	Integer	0,2,4,...,198	Random
odd	Integer	1,3,5,...,199	Random
one-to-one	LV0-T-ROOT (Object-valued)	-	Random
one-to-five	LV0-T-ROOT (Object-valued)	-	Random
one-to-0.1p	LV0-T-ROOT (Object-valued)	-	Random

another randomly selected object which is unique and different than itself.[13] Both *one-to-five()* and *one-to-0.1p()* are multivalued and reference a set of objects.

In our experiments, we used the same database for both the importer and exporter components. Our mechanism requires the importer database to construct a remote type hierarchy (as described in Sect. 4). Hence, the type R-LV0-T-ROOT is created with the same named literal functions as LV0-T-ROOT except that they are actually computed functions which make RPC requests to the exporter. R-LV0-T-ROOT does not contain the object-valued functions [e.g., *one-to-one()* is defined only on LV0-T-ROOT]. This is because the value of a remote object-valued function is undefined at the importer site.[14]

6.2 The queries

We used three different queries to quantify the overhead of our mechanism. The first two queries are used to evaluate the scaling of our instance sharing mechanism (logical horizontal partitioning) as a function of: (1) the fraction of remote instances, and (2) the size of the database. The third query is designed to quantify the overhead of sharing remote stored functions (logical vertical partitioning).

Queries in object-based database management systems generally fall into two categories: associative queries ranging over one or more sets, and navigational queries which traverse the subcomponents of a complex object. The first category of associative queries resembles queries found in traditional relational database systems. Query 1 is used to model this class of queries. It retrieves a fixed percentage of objects that satisfy a certain selection predicate. By varying the percentage of LV0-T-ROOT objects that are remote, we can evaluate the overhead of sharing remote objects. This query retrieves 10% of objects from the database. For example, when the database consists of 100 000 objects (MAXOBJS = 100 000), Query 1 is defined as:

```
select UID(p)
for each LV0-T-ROOT p
where UID(p) < 10 000;
```

[13] A complete explanation of each function can be found in Ghandeharizadeh (1993a)

[14] If this functionality is desired, a surrogate can be created for the remote object resulting from applying the one-to-one()

Navigational queries traverse the subcomponents of a complex object and typically result in expensive random disk accesses. Query 2 is used to evaluate our mechanism when processing this class of queries. We used the *one-to-one()* function to model queries that traverse the subcomponents of a complex object. As in Query 1, we measure the response time of the system as a function of the percentage of remote objects. Once again, this query retrieves 10% of the objects in the database. When MAXOBJS is 100 000, Query 2 is as follows:

```
select UID(p)
for each LV0-T-ROOT p
where UID(one-to-one(p)) < 10 000
```

The last query, Query 3, projects out each function defined on the type LV0-T-ROOT with 100% selectivity. For this query, we vary the number of functions which are remote. First, *UID()* is made remote, then *USTR1()* and finally *SH_USTR2()*.

```
select UID(p), USTR1(p), SH_USTR2(p)
for each LV0-T-ROOT p;
```

6.3 Organization of the experiments

The hardware platform used for these experiments consisted of two workstations connected by Ethernet. In order to systematically characterize the behavior of the system, we conducted our experiments on two separate configurations. The first configuration consisted of an Iris importer and an Omega exporter.[15] The purpose of this configuration was to demonstrate the feasibility of our mechanism and to gain an intuitive understanding of its overhead. The second configuration consisted of Omega as both the importer and exporter. This configuration provided a far more accurate environment for quantifying this overhead of our mechanism, as we had access to Omega's source code and were able to instrument it using simple modifications. We evaluated the performance of each query for three different database sizes (MAXOBJS): 1000, 10 000, and 100 000 objects. In all experiments, each remote object at the importer site were chosen at random from the exporter's database with no object being imported more than once.

[15] The complement configuration of Iris exporter and Omega importer was also evaluated. These results were eliminated because they provided no additional observations

6.4 Heterogeneous configuration

This configuration consisted of an Iris (version DPP 4.0) importer running on a HP 9000/834 workstation and an Omega exporter running on a HP 9000/720 workstation. The goal of this experiment was to demonstrate that the local data manipulation language of Iris was preserved while providing transparent access to the remote objects that resided on the Omega exporter. To this end, the experiment was quite successful. However, since this version of Iris was an early prototype, we had some difficulties creating large databases and making accurate measurements. Nevertheless, the results of running the queries showed that the overhead for remote access was approximately 25%, making remote accesses quite comparable to local access. As we describe in Sect. 6.6, this overhead depends on a number of factors that could not be fully quantified due to our lack of accessibility to the Iris source code. However, our course grain measurements for this configuration indicated that the overhead of our mechanism was relatively low as compared to the local database accesses performed by Iris. The high local access times for Iris resulted from the use of the *latebind()* function. *Latebind()* is a computed function which must search the meta-data every time an attribute of an object is accessed, resulting in a higher access time [see Sect. 5.1 for an explanation of *latebind()* and why it was used in this implementation].

6.5 Homogeneous configuration

In this configuration, Omega components were used as both the importer and exporter, each running on a HP 9000/720 workstation. The sizes of the exporter's database for the 1000, 10 000 and 100 000 object databases were 600 KB, 6 MB and 60 MB, respectively. The size of the importer's database, however, increases as the percentage of remote instances increases because our mechanism requires one surrogate object per imported object. In our implementation, the size of a surrogate object is approximately twice the size of a local instance of LV0-T-ROOT. Consequently, when the importer contains 100% remote objects, the database size is roughly double the size of the exporter at 1 MB, 10 MB and 100 MB, respectively, for the 1000 object, 10 000 object, and 100 000 object databases. However, in general, we expect the remote information contained in the surrogate to be small when compared to the actual size of the object on the exporter. For these experiments, the Omega components were configured with a 4-KB disk page and a 400-KB (100 pages) buffer pool. The buffer pool size was chosen relative to the database size in order to investigate the following two scenarios: the database can become main-memory resident (1000 objects), the database must remain disk resident (100 000 objects). By choosing a larger buffer pool size, we would have had to modify the size of the database appropriately in order to investigate these alternative cases. This would not have changed the final observations because the Omega system scales as a function of both its buffer pool and database sizes (Ghandeharizadeh 1993a).

Table 2. Response times for query 1

	1000 Objects	10 000 Objects	100 000 Objects
0%	0.98 s	14.23 s	171.80 s
10%	3.42 s	59.25 s	983.54 s
20%	4.28 s	92.24 s	1,686.13 s
40%	5.85 s	159.59 s	3,058.80 s
80%	8.91 s	288.63 s	5,727.04 s
100%	10.35 s	346.64 s	7,045.40 s

6.5.1 Query 1

Table 2 shows the response time of Query 1 for the three databases sizes as a function of the percentage of remote objects. For each column, the results demonstrate a relatively linear increase in response time as a function of the percentage of remote objects. For example, in the 100 000 object database column, the increase in response time between 20% to 40% remote objects and 80% to 100% remote objects is roughly 1300 s (i.e., a constant slope). This behavior is not observed for the 1000 object database column due to a higher percentage of buffer pool hits at the exporter as a function of the percentage of remote objects[16], resulting in a lower access time per object. This is due to the relative small size of the database which becomes main memory resident at the exporter site.

With a large database, the exporter's buffer pool hit ratio stabilizes at a fixed percentage for various percentage of remote objects. Figure 7 shows that the buffer pool hit ratio for both the importer and exporter using the 100 000 object database. The importer observes a higher percentage of buffer pool hits because it processes objects sequentially. Since a disk page contains approximately 200 objects, the importer observes 199 buffer pool hits for each disk I/O. Each time the importer processes a surrogate (recall that this is a randomly chosen object from the exporter), it makes a request causing the exporter to perform a random disk page request which has a significantly lower probability of a buffer pool hit.

When there are no remote objects, the response time of the system increases almost linearly as a function of the database size (see the first row in Table 2). However, when there are remote objects, the increase in response time becomes superlinear (a 20-fold increase from the 1000 to 10 000 object database, and a 30-fold increase from the 10 000 to the 100 000 object database).[17] This can be attributed to two factors. The first is due to the significant decrease in exporter buffer pool hit ratios as a function of the database size (see Fig. 8). The second factor is due to the significant difference in seek time at the exporter site for different database sizes. The 100 000 object database occupies more than a hundred times as many tracks as the 1000 object database. Since the seek time is proportional to the square root of the distance traveled by the disk head (Gray 1988), this query spends a longer amount of time performing seeks for the 100 000 object database.

[16] For 10% remote objects, the exporter's buffer pool hit ratio is 81% and increases to 99% for a 100% remote object configuration

[17] The increase between the 1000 and 10 000 object databases varies because of increase in the buffer pool hit ratio on the exporter for the 1000 object database, whereas the exporter hit ratio for the 10 000 object and 100 000 object databases remains constant at 71% and 53%, respectively

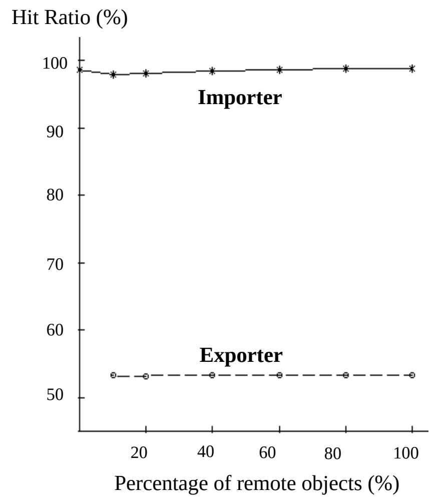

Fig. 7. The 100 000 object database

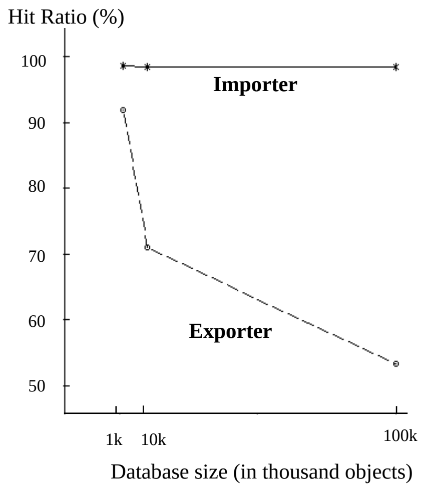

Fig. 8. 40% remote objects

Table 3. Response times for query 2

	1000 Objects	10 000 Objects	100 000 Objects
0%	1.97 s	237.41 s	6,496.11 s
10%	4.44 s	251.05 s	7,165.53 s
20%	5.23 s	301.05 s	7,772.54 s
40%	6.71 s	411.71 s	9,053.62 s
80%	10.04 s	531.11 s	11,374.07 s
100%	11.11 s	563.25 s	12,578.91 s

6.5.2 Query 2

Table 3 shows the response time of the system for Query 2 as a function of the percentage of remote objects for various database sizes. As described in Sect. 6.2, this query navigates the subcomponent of each complex object and results in random I/Os. When there are no remote objects, this query observes a lower percentage of buffer pool hits and a higher disk access time as compared to Query 1. These factors increase the average look up time per object from approximately 1 ms for Query 1 to 30 ms for Query 2, resulting in a significantly higher response time.

In the presence of remote objects, the response time of the system increases modestly as a function of the percentage of remote objects because: (1) the random disk accesses are offloaded to the exporter site, and (2) our implementation of surrogate objects causes the importer to observe a higher percentage of buffer pool hits (see Fig. 9). The additional random I/Os at the exporter site has no impact on its buffer pool hit ratio because all requests made to this site result in random I/Os. The second factor is due to our implementation of a surrogate object which clusters all the information needed to issue a remote request (e.g., R_HOST, R_DBNAME, and R_OID) together in a single tuple on a disk page, causing a higher percentage of buffer pool hits at the importer site.

Table 3 shows that the response time of the system increases superlinearly as a function of the database size. Similar to Query 1, this can be attributed to two factors: (1) the percentage of buffer pool hits decreases drastically as a function of the database size (see Fig. 10), and (2) the exporter incurs a longer seek time as a function of the database size.

Table 4. Response times for query 3

	1000 Objects	10 000 Objects	100 000 Objects
0 Functions	4.36 s	46.09 s	456.75 s
1 Function	13.25 s	382.17 s	6,228.30 s
2 Functions	19.61 s	441.02 s	6,931.03 s
3 Functions	26.94 s	514.28 s	7,487.53 s

6.5.3 Query 3

Table 4 shows the response time of Query 3 for the three database sizes as a function of the remote functions (attributes). In this experiment, the response time increases significantly from no remote functions to one remote function (i.e., compare the first and second rows in Table 4) because the importer performs one RPC for each object in order to access its remote function value. Beyond one remote function, Table 4 exhibits only a modest increase in response time because of a faster service time from the exporter for each additional remote function. The explanation for this is as follows. For each remote function defined on an object and referenced by the query, the importer issues a request to the exporter. The first request typically results in a random I/O at the exporter. However, the exporter observes a buffer pool hit for each additional request (see Fig. 11). This is because all the remote functions are defined on LV0-T-ROOT, and Omega clusters the values of these functions together on a single disk page for each object.

6.6 Discussion and observations

The performance evaluation experiments conducted demonstrate that the overhead of our mechanism depends on: (1) the technique used to resolve overloaded functions at run time, (2) the implementation of a surrogate object, and (3) the network access time. Consider each factor individually. As discussed in Sect. 6.4, the overhead associated with the latebind mechanism of Iris for processing an overloaded function was significant. Our implementation of this construct in Omega exploited the physical organization of the

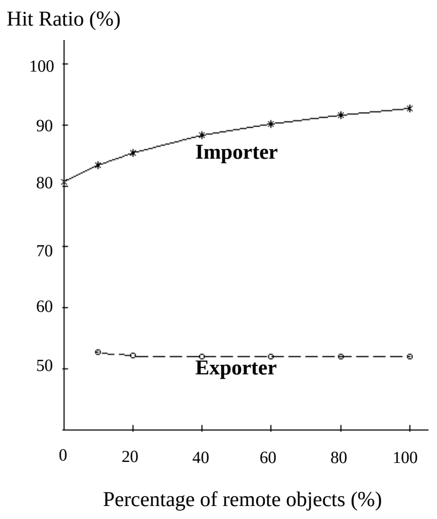

Fig. 9. The 100 000 object database

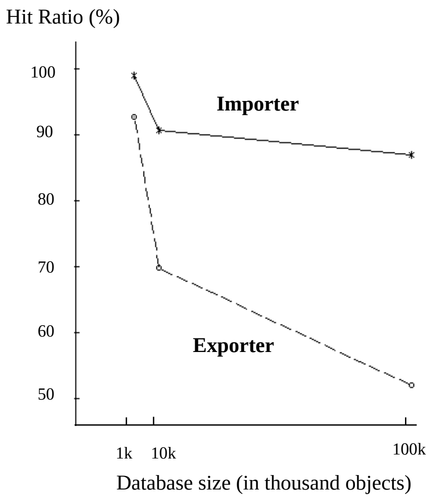

Fig. 10. 40% remote objects

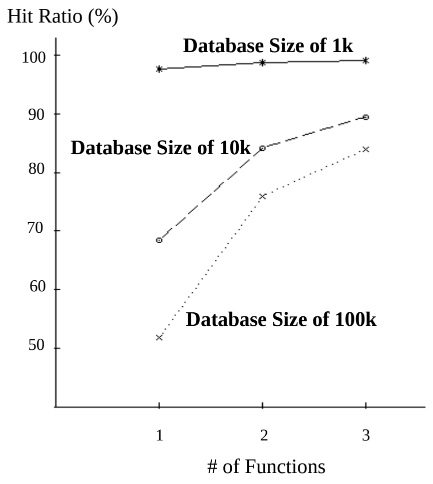

Fig. 11. Exporter hit ratio for Query 3

system to provide a more efficient implementation. This implementation could be further fine tuned in order to minimize its overhead (e.g., replace the repetitious allocate and deallocate calls to the operating system by a scheme which allocates and deallocates memory in larger chunks and minimizes the number of calls to the HP/UX operating system).

Our implementation of the surrogate object clusters the information needed to make a remote request together on a single disk page. Currently, it is 300 B long. Its size could be drastically reduced by: (1) assuming a maximum length for the name of a remote host, its database name, and size of the remote OID, and (2) using an encoding scheme. A smaller surrogate object reduces the disk space requirement of our mechanism at the importer site, and increases the probability of this information becoming memory resident when processing a query.

The network access time consists of: (1) the importer issuing a request, and (2) the exporter returning the results. In all experiments, the importer and exporter were connected using the Ethernet and our network access time was approximately 5 ms per request. This measurement is contingent upon the size of the request and the result (approximately 300 B for the request and 100 B for the result in these experiments). This time could be reduced by using (1) a faster network (FDDI instead of Ethernet), and (2) a more sophisticated protocol that groups requests and submits them at one time (similar to group commit protocol in transaction processing systems), reducing the network traffic.

In addition to these factors, we made the following observations on the overall behavior of the Omega system which implements our mechanism. First, our instance sharing mechanism scales linearly with increasing numbers of remote objects for associative queries over sets (Query 1). Figure 12 shows the time spent in Omega_eval() for Query 1 on both the importer and exporter for the 100 000 object database. The sum of these two times accounts for over 90% of the total response time. The exporter has a larger slope because it performs random disk accesses and observes a lower percentage of buffer pool hits. Second, this mechanism scales sublinearly for navigational queries (Query 2). These queries result in random I/Os at both the exporter and importer. Figure 13 shows that the time spent in Omega_eval() for the importer decreases as the percentage of remote objects increases, due to a smaller working set of active pages and an increased buffer pool hit ratio.

Two factors have a significant impact on these results. The first factor is the implementation of the buffer pool and the choice of buffer pool replacement policy. In our experiments, the Omega components were configured with a 4-KB disk page and a 400-KB (100-page) buffer pool using a least recently used (LRU) replacement policy. Different choices of buffer pool sizes and replacement policies would significantly alter our results. Second, the physical organization of data and techniques employed to retrieve it can impact the results. For example, in the experiments for Query 1, all the imported objects were chosen at random from the exporter. However, in a more probable situation, the imported objects might be members of the same type which would then be clustered together resulting in an overall decrease in the observed response time; another example in our experiments where clustering (or lack of) would have affected

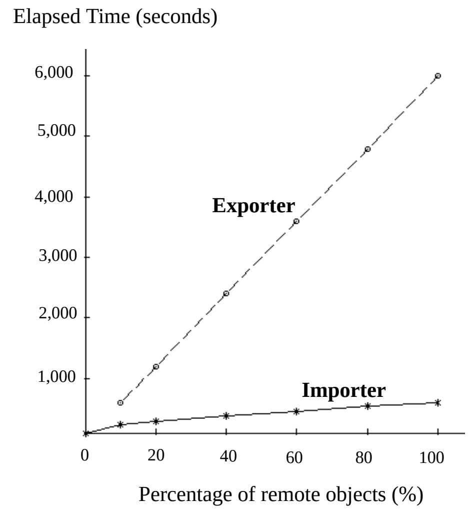

Fig. 12. Omega_eval() times for Query 1

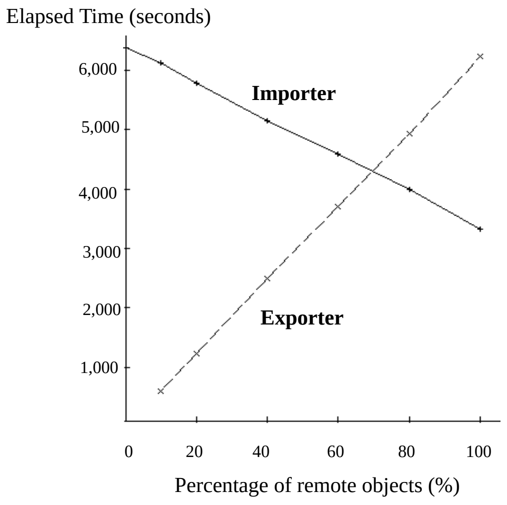

Fig. 13. Omega_eval(t) times for Query 2

our results for Query 3. With this query, the overhead of referencing additional remote functions beyond one remote function was minimal, due to the natural clustering of functions at the exporter site. Clearly, in different component implementations, this intra-object clustering technique may be violated. In general for all the experiments, our Omega components used only the simplest of clustering strategies and assumed no auxiliary access methods.

7 Conclusions and future directions

In this paper, we have described an approach and mechanism for the transparent sharing of objects in an environment of interconnected (networked), autonomous database systems. An experimental prototype implementation has been described, along with an analysis of its performance. Our approach provides a mechanism that allows users to import remote objects directly into their local database environment transparently. The experimental prototype system that has been designed and implemented is based on the Iris and Omega object-based database management systems; this system supports the sharing of data and meta-data objects (information units), as well as units of behavior. The results of experiments we performed to evaluate the performance of our mechanism demonstrate the feasibility of database transparent object sharing, and provides insight into the performance overhead and tradeoffs involved.

There are several aspects to the intended direct and practical impact of our work:

– *Existing components.* Throughout this work we have paid careful attention to the requirement that there should be no modification to existing database management system software. As a result, our approach requires no modification to the query processor or any other component of the local system. In particular, we do not assume a standard global OID space of which each component must be aware. Not only does our approach support the existing database management system software (e.g., the query language), but it also supports existing application programs developed by users.

– *Sharing patterns.* Our approach introduces new sharing patterns not found in other systems. We support the sharing of objects at various levels of granularity and abstraction. In addition, we support the sharing of behavioral objects. These sharing patterns may be established dynamically from multiple sources and are determined individually by each component; there is no global schema.

– *Database transparency.* Another feature of our approach is database transparency. With this level of transparency, users are more productive, since learning a new language or moving to a new environment is not a prerequisite to sharing information.

– *Decoupling of data and behavior.* We have considered the importance of decoupling the location of (persistent) data and the location of the functions that operate on data in a distributed environment. Traditional approaches inextricably link the location of the data and the execution of the operation. Our implementation provides an approach and callback mechanism which addressed the problems associated with possible side-effects of behavioral objects.

– *Common data model.* The results described here may also impact the area of heterogeneous database systems with respect to data model and conceptual schema heterogeneity. One of the critical factors that determine the success of these systems is the choice of a common data model used for integrating component database systems (Sheth 1990). We expect that our experiences with a functional object-based data model provide insight into the design and functionality of future CDMs.

It is important to note that the experiments in the evaluation portion of the research were designed to quantify the overhead of our mechanism in a standard environment. There are several ways of improving the performance of our mechanisms. First, there is an inherent parallelism that exists in a federated environment. All of our experiments consisted of only one exporter. In the presence of multiple exporting components, asynchronous or multi-threaded RPC requests can be used to retrieve the results from these components in parallel. Second, the importer can cache the remote objects in order to minimize the number of RPC requests. Access to a remote object is always more expensive than access to a local object. Hence, if a remote object is "cached" in the local database as a local object, its access times will decrease. By providing a generalized mechanism for resolving overloaded functions (such as the one implemented using Omega), a local stored function can be used to locally cache the value of the computed function which accesses the remote instance's state. This provides the basis for a simple but powerful mechanism for selectively caching the remote state of an imported object. The strategy for setting and updating the values of the stored local functions is determined by an appropriate caching policy. We are in the process of studying the impact of various caching policies for this high-level object caching scheme in order to reduce the amount of remote accesses.

Acknowledgements. We would like to gratefully acknowledge the insights and helpful comments of the anonymous referees in improving the context and presentation of this article. We are also indebted to Joachim Hammer and Antonio Si (of USC), for their helpful contributions and comments on the content of the article. This research was supported in part by NIH/NIMH grant 1-P20-MH/DA52194-01A1, NSF grant IRI-9110522, NSF NYI award IRI-9258362, and NSF infrastructure grant CDA-9216321.

References

Afsarmanesh H, McLeod D (1989) The 3DIS: An Extensible, object-oriented information management environment. ACM Trans Office Inf Syst 7:339–377

Anderson TL, Berre AH, Mallison M, Porter H, Schneider B (1990) The hypermodel benchmark. In: Proceedings of the 2nd International Conference on Extending Data Base Technology, March

Atkinson M, et al (1989) The object-oriented database system manifesto. In: Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, December

Birrell A, Nelson B (1984) Implementing remote procedure calls. ACM Trans Comput Syst 2:39–59

Cattell R (1988) Object-oriented DBMS performance measurement. In: Advances in object-oriented databases: Proceedings of the 2nd International Workshop on Object-Oriented Database Systems. (Lecture Notes in Computer Science, vol 334). Springer, Berlin Heidelberg New York

Chou HT, DeWitt DJ, Katz R, Klug T (1985) Design and implementation of the Wisconsin storage system. Software Pract Exper 15

Fang D, McLeod D (1992b) A Testbed and mechanism for object-based sharing in federated database systems. Technical Report USC-CS-92-507, Computer Science Department, University of Southern California, Los Angeles CA 90089-0781

Fang D, Hammer J, McLeod D (1992a) An approach to behavior sharing in federated database systems. In: Distributed object management. Morgan Kaufman, San Mateo, Calif

Fishman D, Beech D, Cate H, Chow E, Connors T, Davis T, Derrett N, Hoch C, Kent W, Lyngbaek P, Mahbod B, Neimat M, Ryan T, Shan M (1987) Iris: an object-oriented database management system. ACM Trans Office Inform Syst 5:48–69

Gardarin G, Gannouni S, Finance B, Fankhauser P, Klas W, Pastre D, Legoff R (1995) IRO-DB : A distributed system federating object and relational databases. Prentice-Hall, Englewood Cliffs, NJ

Ghandeharizadeh S, Choi V, Bock G (1993a) Benchmarking object-based constructs. In: Proceedings of eight brazilian symposium on databases

Ghandeharizadeh S, Choi V, Ker C, Lin K (1993b) Omega: a parallel object-based system. In: Proceedings of the fourth australian database conference

Gray J, Sammer H, Whitford S (1988) Shortest seek vs shortest service time scheduling of mirrored disks. Tandem Comput

Hammer J, McLeod D (1993) An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. Int J of Intell Coop Inf Syst 2:51–83

Hammer J, McLeod D, Si A (1994) An intelligent system for identifying and integrating non-local objects in federated database systems. In: 27^{th} Hawaii International Conference on System Sciences, pp 398–407

Heimbigner D, McLeod D (1985) A federated architecture for information systems. ACM Trans Off Inf Syst 3:253–278

Hull R, King R (1987) Semantic database modeling: survey, applications, and research issues. ACM Comput Surv 19:201–260

Kent W, Ahmed R, Albert J, Ketabchi M, Shan M (1993) Object identification in multidatabase systems. In: Interoperable database systems (DS-5). North Holland, Amsterdam

Kim W, Banerjee J, Chou HT, Garza JF, Woelk D (1987) Composite object support in an object-oriented database system. In: Proceedings of the conference on object-oriented programming systems, languages, and applications, pp 118–125

Lecluse C, Richard P, Velez F (1988) O2, an object-oriented data model. In: Proceedings of the ACM SIGMOD International conference on management of Data. ACM SIGMOD, June

Litwin W, Abdellatif A (1986) Multidatabase interoperability. IEEE Comput 19

Maier D, Stein J, Otis A, Purdy A (1986) Development of an object-oriented DBMS. In: Proceedings of the conference on object-oriented programming systems, languages, and applications, pp 472–482. ACM

Object Management Group (1991) The common object request broker: architecture and specification. Document no. 93-12-14. P;G

OMG Document Number 91.11.1 (1992) Object management architecture guide, OMG

Shan M (1989) Unified access in a heterogenous information environment. IEEE Off Knowl Eng 3:35–42

Sheth A, Larson J (1990) Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput Surv 22:183–236

Shipman D (1981) The functional data model and the data language DAPLEX. ACM Trans Database Syst 2:140–173

Smith J, Bernstein P, Dayal U, Goodman N, Landers T, Lin K, Wong E (1981) Multibase: integrating heterogeneous distributed database systems. In: Proceedings of the National Computer Conference. AFIPS, June. pp 487–499

Stonebraker M, Neuhold E (1988) A distributed database version of INGRES. In: Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks, University of California, Berkeley, May, pp 19–36

Sun Microsystems (1988) Network programming guide. Sun Microsystems

Templeton T, et al (1987) Mermaid: A front–end to distributed heterogenous databases. In: Proceedings of IEEE, pp 695–708

Williams R (1981) R*: An Overview of the architecture. (Technical Report RJ3325) IBM