Plug and Play: Interoperability in CONCERT

Lukas Relly and Uwe Röhm

Database Research Group, Institute of Information Systems,
ETH Zentrum, 8092 Zürich, Switzerland
{relly|roehm}@inf.ethz.ch
http://www-dbs.inf.ethz.ch/

Abstract. In order to make database systems interoperate with systems beyond traditional application areas a new paradigm called "exporting database functionality" as a radical departure from traditional thinking has been proposed in research and development. Traditionally, all data is loaded into and owned by the database, whereas according to the new paradigm data may reside outside the database in external repositories or archives. Nevertheless, database functionality, such as query processing, and indexing, is provided exploiting interoperability of the DBMS with the external repositories. Obviously, there is an overhead involved having the DBMS interoperate with external repositories instead of a priori loading all data into the DBMS. In this paper we discuss alternatives for interoperability at different levels of abstraction, and we report on evaluations performed using the CONCERT prototype system making these cost factors explicit.

1 Introduction

Todays Database Management Systems (DBMS) make the implicit assumption that their services are provided only to data stored inside the database. All data has to be imported into and being "owned" by the DBMS in a format determined by the DBMS. Traditional database applications such as banking usually meet this assumption. These applications are well supported by the DBMS data model, its query and data manipulation language and its transaction management. Advanced applications such as GIS, CAD, PPC, or document management systems however differ in many respects from traditional database applications. Individual operations in these applications are much more complex and not easily expressible in existing query languages. Powerful specialized systems, tools and algorithms exist for a large variety of tasks in every field of advanced applications requiring these systems to interoperate and make their data available to other systems.

Because of the increasing importance of advanced applications, DBMS developers have implemented better support in their systems for a broader range of applications. Binary Large Objects provide a kind of low-level access to data and allow individual data objects to become almost unlimited in size. Instead of storing large data objects in BLOB's, some newer systems such as ORACLE

(Version 8) and Informix (Dynamic Server with Universal Data Option) provide the BLOB interface also to regular operating system files. Because the large objects in any of these two options are uninterpreted, database functionality for this kind of data is only very limited. In order to better support advanced applications, the standardization effort of SQL3 specifies, among others, new data types and new type constructors. Most recently, SQL3 and object-orientation have fostered the development of generic extensions called datablades [inf], cartridges [ora], and extenders [ibm]. They are based on the concept of abstract data types and often come with specialized indexing.

Although they provide better support for advanced applications, however, except for the file system case, they all have the same fundamental deficiencies: First, it is the DBMS together with its added extensions that prescribes the data structure and data format of the data to be managed. The consequence is that all complex specialized application systems and tools must be rewritten using the data structures enforced by the DBMS, or at least complex transformations must take place to map the DBMS representation into the application representation. Second, the DBMS owns the data. All data has to be physically stored inside the DBMS requiring to possibly load gigabytes of data into the database store.

These observations led to a radical departure from traditional thinking as it is expressed in [SZB+96]. In the CONCERT project at ETH, we focus on *exporting database functionality* by making it available to advanced applications instead of requiring the applications to be brought to the DBMS. In [RSHN97] and [RSS98], we presented the concepts needed to enable the DBMS to interoperate with external data repositories exporting its functionality to data stored outside the DBMS. Query processing and indexing is performed by generic methods of physical database design invoking operations of user-defined abstract (external) data types. In this paper, we identify different levels of abstraction, at which interoperability for query processing can take place, and we present performance measurements identifying the costs required for the additional flexibility. With the exception of [SW93] we are not aware of other work that deals with external data and related performance measurements.

This paper is organized as follows: Section 2 introduces the two possible levels of abstraction, at which interoperability of the CONCERT query engine with external repositories can take place. Section 3 discusses the lower level of abstraction exploiting the integration of abstract data objects in the Database Kernel. In Section 4 we present CONCERT's object manager "Harmony", which adds higher level query processing capabilities. Section 5 concludes.

2 The CONCERT Architecture

In CONCERT, interoperability can take place at two different levels of abstraction corresponding to the CONCERT system architecture. CONCERT consists of a database kernel system and a generic object manager. Figure 1 gives an overview of the CONCERT architecture. Interoperability at the kernel level is performed

on an object by object basis while interoperability at the object manager level is based on accessing collections of objects.

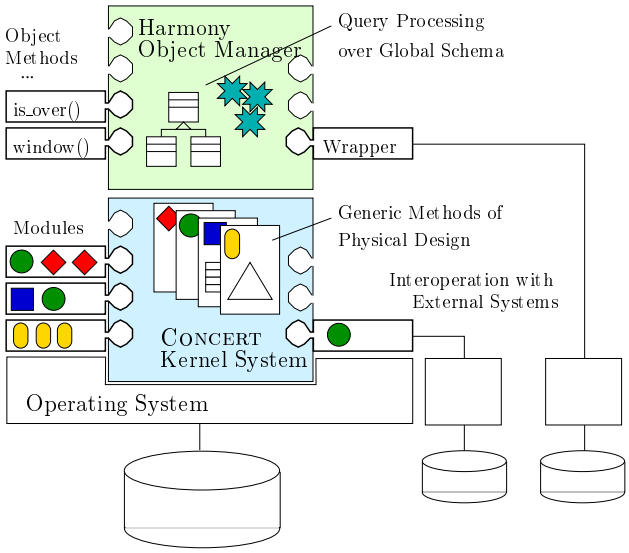

Fig. 1. Overview of the CONCERT system architecture

The CONCERT kernel system provides low-level database functionality such as storage management, low-level transaction management, and predicate and projection evaluation on single collections as basic query processing capability. It's role is comparable to System R's RSS [ABC+76], the Starburst Kernel [LMP87] or the DASDBS Kernel [SPSW90]. In order to make data management efficient, the kernel is *tightly bound to the underlying operating system* providing multithreading and exploiting efficient secondary storage access using a memory-mapped buffer [BKRS94]. With respect to interoperability, the kernel's capability exporting physical database design making it available for external systems is important. The kernel implements *generic methods of physical design* such as a Btree index, an inverted file index, an Rtree-like spatial index. In contrast to traditional systems, these indexes are not connected to a predefined type system, rather they rely on object properties. These object properties are made available through *modules* that can be plugged into the kernel providing access to external objects through method invocation. The CONCERT kernel and its interoperability evaluation is presented in Section 3.

The Harmony object manager sits on top of CONCERT's kernel system. At this level, CONCERT abstracts from single data objects. Harmony provides declarative access to collections of abstract objects according to the *global schema*. The main concept of this layer is a collection, which represents a homogeneous set of data objects. Its elements are accessed through an iterator interface. Fur-

thermore, Harmony defines a query algebra over the collections. Each collection of the object manager is in fact a queryable collection: it offers an `evaluate` method, through which a *declarative query* can be executed on its elements resulting in a new collection representing the result set. The global data model is object-oriented. The user can define *object methods*, whose object code is plugged into the object manager. Further, Harmony can interoperate with external systems through *wrappers*. They allow to export the query processing functionality of CONCERT to external repositories. The Harmony object manager and its interoperability evaluation is presented in Section 4.

Example: To illustrate the interplay of the different layers of CONCERT, we look at the following simple example: Imagine a geospatial image archive. A satellite periodically generates new images together with descriptive information (e.g., current date and time, satellite position, etc.). These satellite images are stored in a huge tape archive. Storing the image triggers the creation of a corresponding new image object in CONCERT, which holds the meta information about the image and its position in the tape archive – but without the image data itself.

First, we ask for the titles of all images. This query will be mainly processed inside the CONCERT kernel. The abstract image objects are of the plugged-in type `SatelliteData` (see Appendix B), which has the general form `RECORD ( SCALAR, SPATIAL, SPATIAL )`. Harmony scans the image objects, asking for their first component, which the global schema declares as `title` attribute (see Appendix C). The kernel calls the concept-typical method `SUB_OBJECT()` on the corresponding plugged-in type `SatelliteData`, and returns the extracted subobject. This is in fact a `string`. Harmony generates the result collection of string values. No further processing is needed.

Second, we want to display a part of an image, whose title we know. Harmony now scans the image objects, asking for all such objects, where the first component holds the specified string-value (i.e., is the search title). This can again be evaluated inside the CONCERT kernel. On the retrieved abstract `SatelliteData` Harmony now executes the user-defined `window()` method. This method extracts the part of the image we want to display. In order to do so, it falls back on the concept-typical methods `SPLIT()` and `COMPOSE()` of the SPATIAL component of each `SatelliteData`. These methods are provided by the kernel module `SatelliteImage`. They retrieve the needed parts of the satellite image from the external tape archive. This is an example for interoperability at the storage system layer.

Third and last, there might exist a web page, on which weather data about Europe is published. Now we are interested in all satellite images of Europe, which are taken at a the same time, as this published weather information. This means, we need interoperability at the object manager layer. Harmony has to access the external web server via a corresponding wrapper. This wrapper transforms the contents of the page into a collection of weather data entries. Harmony afterwards joins these entries with the image objects stored inside CONCERT according to the time attributes.

3 An Abstract-Object Kernel System

Under the term "Object-Relational", database systems have become popular, that allow extensions to their kernel system. Such extensions are called blades, cartridges, or alike. They allow the DBMS to be extended by *application specific types* and *access methods*. While implementing new types is relatively easy, new access methods is not. The new access method has to cooperate with the various components of the DBMS, such as concurrency control, data allocation, query evaluation, and optimization. This requires substantial knowledge of the DBMS internals. In contrast, the CONCERT kernel offers a limited, built-in set of physical design mechanisms in form of generic, trusted DBMS code provided by the DBMS implementor. Physical design is performed through relating new types to the fundamental concepts of the built-in physical design mechanisms.

3.1 Concepts of Physical Database Design

In [Sto86], Stonebraker introduced the idea of a generic B-Tree that depends only on the existence of an ordering operation to index arbitrary data objects. Our CONCERT approach generalizes this idea by identifying *all relevant concepts of physical database design* and expressing them by the so called *concept typical* operations required to implement them over external data. The data objects are treated as abstract data types (ADT) in CONCERT, and physical database design is performed based on the operations of the ADT only. These ADT's are user-defined and their methods are dynamically linked to the kernel at run time. In order to implement search tree access methods, a generic search tree approach (similar to GiST [HNP95]) can be used as it integrates nicely into the CONCERT framework.

The physical design concept behind Stonebraker's generic B-Tree is that of data objects having a *scalar* property. Therefore, we call it the `SCALAR` concept and its concept typical *ordering operation* `COMPARE`. The comparison operation is sufficient to instantiate a generic B-Tree index without any further knowledge of the data objects. A second concept called `RECORD` concept allows to identify components of objects. A data object might be decomposed into object parts. This is exploited for example in a relational context as vertical partitioning. Its concept typical operation is the decomposition of objects into object parts called `SUB_OBJECT`. A third fundamental concept of physical database design is the one found for example in the information retrieval context, where objects are organized according to sets of object properties such as the index terms of the document object. The concept typical operations of this concept are the ones iterating over the set of properties. We therefore call it the `LIST` concept. The iteration allows the properties to be entered for example into an inverted file index. Finally, the last concept of physical database design, we identified in CONCERT is the one concerned with spatially extended objects and is therefore called the `SPATIAL` concept. It is used for expressing space–subspace relationships as they appear in GIS and CAD systems, but also in temporal applications.

These concepts are the means for interoperability at the CONCERT kernel level as discussed in Section 2. From a physical design point of view, storing the satellite images of our sample application in a tape archive while storing the corresponding metadata in the CONCERT storage component corresponds to a vertical partition of the image objects. The concept typical operation required is the decomposition of the `RECORD` concept. Therefore, the Application Programmer can make the fact of the image residing in the tape archive known to the kernel via the `RECORD` concept by implementing the concept typical operation `SUB_OBJECT`. This operation is responsible for accessing the image on the archive.

In addition to the four concepts `SCALAR`, `RECORD`, `LIST` and `SPATIAL` and their concept typical operations, three fundamental operations are required for all abstract objects. They are needed to pass abstract objects across system internal interfaces. If for example an abstract object is inserted into a Btree index, the object has to be recursively passed through the nodes of the tree. The kernel has to be able to `COPY` an object, which is the first and most important operation, that any object in CONCERT has to provide. Depending on the usage of the object, copying can be performed in different ways. If the object is to be passed to a function call within the same process, a shallow copy might be appropriate. If the object has to be stored in a database disk page, a full copy is required. In addition, this copy has to be linearized into a single continuous address space. CONCERT allows the copy operation to be driven by a set of copy flags making such distinctions. While the copy operation is specific for each object type, memory allocation has to be performed by the generic database code. In order to get to know the resource requirements, the `COPY_SIZE` operation has to be provided by each object enabling the generic kernel to perform the necessary allocations. In order to actually perform a copy operation, additional resources such as temporary memory, network connections, file handles or alike might possibly be required. These resources have to be freed once the copy is no longer needed by the database. Therefore, the third operation required for all generic objects is the `DELETE_AUX` operation. As a consequence, the usual steps passing abstract objects around is performed as shown in Figure 2.

```
s     := COPY_SIZE (o, copy_flags);
new_o := allocate (s);
COPY (o, new_o, copy_flags);
.. do something with  new_o ..
DELETE_AUX (new_o);
```

Fig. 2. Steps required to move an object around

Appendix A shows the interface definition of the CONCERT kernel concepts and their concept typical operations. It is beyond the scope of this paper to give full details here. More information on CONCERT concepts in particular and the

CONCERT kernel system in general can be found in [BKRS94,BRS96,RSHN97] and [RSS98].

3.2 Performance Evaluations for Interoperability at Kernel Level

It is clear, that the interoperability flexibility available in the CONCERT kernel allowing the kernel to access data from remote repositories has its price. This is not specific for CONCERT but rather inherent to any interoperable system. Because interoperability in the kernel is done at a very low level of abstraction, it is very efficient and measuring the local overhead therefore gives a minimal lower bound for the overhead to be expected in any interoperable system.

Passing objects as parameters of procedure calls accross kernel modules is often used in kernel systems. Therefore, the main reason for the low-level interoperability cost in CONCERT is the fact, that the algorithm in Figure 2 involving method calls to abstract object is executed frequently. In systems with hard-coded object types, passing them as parameters of procedure calls can be specialized for the supported types and therefore can be coded more efficiently. Measuring the overhead of the generic algorithm compared with the hard-coded gives a good indication of the low-level interoperability cost.

We identify three typical cases for base type objects:

- The object type is a built-in type (such as `longint` or `float`) of the compiler, that the database system is compiled with. Copying the object can be done using compiler-generated object assignment code. This is the best possible case for a hard-coded system and the most advantage compared with the generic case is expected. However, for these types, data independence can not be guaranteed, as the type representation is compiler and hardware dependent.
- The object type is compiler independent, but of simple structure and of fixed size. Most standard database types in traditional database systems are of this category, such as `INTEGER`, `NUMBER`, `CHAR(n)`. In some systems, aggregations of simple types such as `ROW` types in SQL3 fall into this category as well. Because their size is known a priori, and their representation is a continuous byte sequence, copying these objects corresponds to a simple `memcopy` operation.
- The object type is of variable size, such as `VARCHAR`, `BLOB` or aggregations of object types. Their object size varies from object to object. Therefore, the object size has to be determined at run time and space allocation has to be performed dynamically.

We do not discuss object type with complex structure, because there is virtually no cost difference between hard-coded and abstract objects.

Using the CONCERT kernel system we measured the three typical cases comparing hard-coded with abstract objects using generic algorithms. Figure 3 summarizes the results showing the copy time in nanoseconds on two different SUN Solaris system architectures.

	SparcCenter 2000		UltraSparc 1	
	hard-coded	generic	hard-coded	generic
int (32bit)	446	2108	153	840
NUMBER	1345	2877	719	1389
VARCHAR	11363	13199	3539	4196

Fig. 3. Comparison copying hard-coded objects versus abstract objects using generic algorithms (execution time in ns)

It does not surprise that moving more complex objects around is much more expensive than simple, small ones. It is clear that the more recent system architecture (UltraSparc 1) is substantially faster than the older one. From these measurements, we see that the interoperability costs, that is the difference between the hard-coded version and the generic version, is especially high for very simple objects (with an interoperability overhead of much more than 100%). The overhead is much smaller for larger objects (the VARCHAR object had an average length of 150 characters resulting in an overhead of approximately 15%). For even larger and more complex objects, the overhead is only a few percent. While the first case due to the of lack of data independence is not very relevant for database interoperability, already in the second case the overhead is in the order of improvement of one hardware generation. Furthermore, the local interoperability cost for small objects is very small compared with the overall system cost. We conclude that *building a system capable of low-level interoperability using generic instead of hard-wired algorithms has only a minimal impact on the overall system performance: the rapid hardware development makes low-level interoperability affordable.* In the next section we concentrate on the higher level aspect of interoperability.

4 An Abstract-Object Query System

The usual understanding of interoperability between systems is quite narrow, addressing only SQL-interoperability. Commercial "SQL middleware" like the Informix Enterprise Gateway Manager, Oracle Transparent Gateways, or Sybase' OmniSQL Server rely on a declarative surface of object managers. This means, they presume a declarative interface like ODBC or JDBC, which is already capable of executing SQL. A precondition, which is not feasible for non-database systems. Therefore, we will not discuss this, in fact, third level of interoperability in this paper.

We rather address interoperability of the layer between the kernel system and the (declarative) user interface. CONCERT's object manager only relies on single-scan collection interfaces. Especially, it does not require any further query capabilities of the storage system. However, if certain repositories provide such capabilities, they can be exploited. The object manager itself adds complete query functionality to the underlying storage kernel and exports a declarative interface to subsequent system layers.

An additional important feature of CONCERT's object manager is its distributed peer-to-peer query execution. While the kernel offers typical storage system functionality for a single node system, the object manager layer is capable of distributing its query execution among several sites with CONCERT instances. As underlying infrastructure the CORBA middleware standard [Obj95] is used. CORBA specifies a communication infrastructure for arbitrary heterogeneous components of distributed systems. Furthermore, it defines a set of basic system services and standard components. One system service is of particular interest for CONCERT: the *Object Query Service (OQS)*. As core of CONCERT's object manager, we have designed and realized an implementation of the OQS, called *Harmony*.

4.1 The Harmony Query Service

Harmony's query algebra operators are implemented by means of physical query operations. E.g., the *join* operator might actually be executed as simple *nested loop join*, or more sophisticated algorithms like a *merge join* or *bind join*. Harmony evaluates a query by a sequence of such physical query operations which are partially ordered: an *execution plan of the query*. An execution plan has a tree structure, each node representing one physical operation. Edges connect the nodes according to their partial ordering. Each edge therefore represents the dataflow between subsequent operations. The leaves stand for the data sources needed for query evaluation. The root node represents the result of the query.

Fig. 4. Example query and corresponding execution plan in Harmony.

An efficient implementation of the OQS is not straightforward. To come up with a realization of a CORBA-based query service whose performance is competitive, we have deployed classical database concepts, notably dataflow evaluation, bulk-transfer and intra-query parallelism: internally, Harmony evaluates a query in a dataflow manner according to an execution plan. As introduced in Section 2, each operation of the plan corresponds to a *QueryableCollection* object instantiated before evaluating the query. These objects actually implement the query algebra operators of Harmony. In addition to the usual operators, the query algebra includes *meta operators*: *wrap*, *send* and *receive*. Meta operators do not change the data stream, but perform some control function.

Send / Receive The *send/receive* operators model the asynchronous set-oriented data transfer between sub-plans. They are used for performance optimizations of distributed query processing and of interoperating queries, as we will show in Subsection 4.3. For a more in-detail description and evaluation of send/receive see [RB98].

Wrap wrap (ω) abstracts from the underlying storage systems and therefore allows to integrate different storage systems on the logical level. In order to interoperate with a certain external system, an instance of the *wrap* operator for this storage system must be created. These instances correspond to the concept of *wrappers* [GMHI+95]. Available wrappers are configured via Harmony's global schema (cf. Appendix C). Their code is dynamically loaded at runtime into Harmony as soon as the first access to a collection at the corresponding repository is required.

A Harmony wrapper transforms the interface of the underlying storage system into the collection/iterator interface needed by Harmony for query evaluation. This includes type conversion into the CORBA type system. As Harmony provides all further query operations, a simple wrapper does not need to implement own query functionality. The only functionality a wrapper must provide is to access data collections of its source. In the simplest case, this means the ability to scan the source. A wrapper of a more sophisticated system may publish its query capabilities. Harmony can then decide to delegate whole sub-queries to such a wrapper, which will translate it into and execute it exploiting the local query language.

We distinguish three different types of repositories which can be accessed by Harmony:

- The CONCERT kernel storage system. This is the internal interface to CONCERT's own kernel system. The major aspect here is the integration of the kernel's abstract object types into the CORBA-based query service. This affects the type system and the user-defined methods, as we indicated in the satellite image example before. A further discussion follows in Subsection 4.2.
- A data repository. As mentioned in the introduction, the main idea of CONCERT is to export database functionality to external systems. At the level of the object manager this is possible by providing a wrapper for such a simple repository, e.g., an GIS system or a file system. Via the wrapper, CONCERT is capable of evaluating declarative queries on data managed by such repositories. Furthermore, externally stored data can be combined (joined) with objects managed by CONCERT itself (as shown with the third example query of Section 2). A similar approach is taken by Microsoft with OLE DB [Bla96,Rau96], but only for relational data and without an own storage system.
- A database system. A special case of interoperability with external storage system is the integration of full-fledged database systems. First, such systems have own query capabilities which should be exploited by the corresponding

wrapper. Second, the interface level at which Harmony now interoperates with the external system, is typically one above the layer of the object manager. E.g., with a relational database, the wrapper must use the Embedded SQL API, which is already a declarative interface. As state-of-the-art database system do not provide lower APIs, interoperability is simply not possible at a lower abstraction layer.

4.2 Abstract-Object Types in Harmony

As Harmony relies on the CORBA standard, it has to type the data with respect to the CORBA type system. Here, we have to distinguish two different abstraction levels of data access in Harmony. So far, we concentrated on the higher level of collections of objects. All data access and query capabilities of Harmony are defined with respect to abstract (queryable) collection and iterator interfaces. These are CORBA objects, defined in CORBA IDL and fully embedded in the middleware infrastructure.

At the abstraction level below, we are interested in the member objects of Harmony's queryable collections. As they may be of arbitrary type, Harmony employs the dynamic typing capability of CORBA, its *any* type. The actual content type of an *any* value is determined at runtime. For example, the data produced by the ω_{RDBMS} node in Figure 4 corresponds to tuples of the form `(Description VARCHAR(100))`. The wrap operation maps these data items into the CORBA type `sequence<any>`. At runtime, it contains values of type `string`.

In order to benefit from the abstract object types of the kernel, we extended CORBA's own dynamic type to include CONCERT's build-in types. For the CORBA system this is a user-defined *opaque* type, which Harmony provides the code for. This means, Harmony can also exploit the capabilities of concept typical operations. Especially, this allows to combine interoperability at the physical and the logical level. The Harmony object manager can evaluate a query over collections of CONCERT objects, which are actually stored outside the kernel and accessed by the concept typical operations.

4.3 Performance Evaluations for Interoperability at Object Manager Level

In this subsection we are interested in quantifying the costs of interoperability. It is clear, that any access through the wrapper interface of Harmony must be more expensive than direct access to the underlying storage system. Interoperability does not come for free.

Beside the performance difference of a three tier (client – Harmony– repository) against a two tier system approach (client – repository), another major performance bottleneck must be considered: as mentioned before, the Harmony object manager is capable of distributed execution of an query by exploiting a CORBA infrastructure. As any middleware system, CORBA introduces some overhead in order to provide the location and implementation transparency one

expects from such a system. E.g., it has to marshal/demarshal each value transmitted via the middleware. This additional data conversions certainly affect the performance of a middleware-based system.

The answer Harmony offers for this issue are its *send/receive* meta operators introduced above. These operators allow for intra-query parallelism and optimized bulk-data transfer between different query operations. While the receive operator consumes the set of intermediate results it got from the corresponding set operator, the subsequent query operators can produce the next partial result in parallel. For further details see [RB98].

As a starting point to quantify the interoperability overhead of Harmony, we measured the access times for data access to a result set of 30000 string values stored in a relational database. We chose a database system, so that we could easily obtain a reference result. Therefore, we executed the SQL query shown in Figure 4 via an Embedded SQL/C program. The query was executed in two different system configurations: with the client and server on the same local machine and separated via a local area network (LAN). The results are compared to the execution times of Harmony. First, we submitted the same query to the RDBMS via Harmony's RDBMS wrapper and retrieved the result set. Second, we introduced a send/receive pair above the wrap-operation, so that the external database produced its result items concurrently to the further processing in Harmony. For our experiments, the Harmony client retrieved the query results in chunks of 100 values. This is the same number of items as the ESQL/C program retrieved with one of its array fetches. The runtimes are presented in Figure 5.

	RDBMS	**Harmony**	
		wrap only	with send/receive
local	26.22s	391.63s	32.04s
LAN	24.19s	366.45s	29.98s

Fig. 5. Local/remote data access times (seconds) for CONCERT and a RDBMS.

This experiment clearly shows a massive interoperability overhead for the "naive" execution of the query with Harmony. Without deploying any query optimization techniques, data access through the Harmony wrappers is about 15 times slower than direct access to the RDBMS (first two columns in Figure 5). This drastically changes, if we introduce intra-query parallelism in Harmony: data access through Harmony is now only about 20% slower as a direct database access via Embedded SQL/C. This result shows, that *medium-level interoperability is affordable, as long as it is combined with proven query optimization techniques.*

5 Conclusions

In this paper, we showed, how interoperability between heterogeneous storage systems is achieved in our CONCERT system. We discussed two different levels of interoperability: between kernel systems, and on the object manager level. The main motivation is to be able to export database functionality to external systems.

This is achieved in a very flexible way: the kernel allows to plug-in modules with user-defined types, which only have to provide the fundamental concept-typical operations. Above the kernel, the Harmony object manager includes a wrap-operator in its query algebra. Instances of this operator, so-called wrappers, allow to interoperate with external repositories at the query-processing level. Their only required functionality is a simple-scan interface. Extensions at all levels are implemented in a plug and play like fashion exploiting the operating system capability of dynamic linking.

We also presented first results of a cost evaluation of interoperability with CONCERT at the different levels. From our experiments, we can learn two conclusions: First, the interoperability overhead increases with the abstraction layer at which interoperability takes place. Second, the overhead can be minimized by exploiting well-known database optimization techniques, like bulk-transfer and intra-query parallelism in the case of Harmony. Naive interoperability is very costly. An efficient implementation of an interoperability interface needs careful design and conscious deployment of proven database techniques.

The ideas presented here are part of ongoing work. We are primarily interested in further evaluating the interoperability costs at the different abstraction levels, especially in comparison with spatial data access modules of commercial systems. We also plan to investigate, how external data can be accessed fully dynamically via wrappers without prescribing the data format.

References

[ABC+76] M. M. Astrahan, M. W. Blasgen, et.al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[BKRS94] Stephen Blott, Helmut Kaufmann, Lukas Relly, and Hans-Jörg Schek. Buffering long externally-defined objects. In *Proceedings of the Sixth International Workshop on Persistent Object Systems (POS6)*, pages 40–53, Tarascon, France, September 1994. British Computer Society, Springer-Verlag.

[Bla96] Jose A. Blakeley. Data access for the masses through OLE DB. In *Proceedings of the 25th ACM SIGMOD Conference on Management of Data, Montreal, Canada*, June 1996.

[BRS96] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An open abstract-object storage system. In *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, June 1996.

[GMHI+95] H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information

sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 562–573, September 1995.

[ibm] Ibm corporation. `http://eyp.stllab.ibm.com/t3/`.

[inf] Informix corporation. `http://www.informix.com/informix/products/-options/udo/datablade/dbmodule/index.html`.

[LMP87] Bruce Lindsay, John McPherson, and Hamid Pirahesh. A data management extension architecture. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD Record, pages 220–226. IBM Almaden Research Center, San Jose, 1987.

[Obj95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[ora] Oracle corporation. `http://www.oracle.com/st/cartridges/`.

[Rau96] Stephen Rauch. Talk to any database the COM way using the OLE DB interface. *Microsoft Systems Journal*, July 1996.

[RB98] Uwe Röhm and Klemens Böhm. Working together in harmony — an implementation of the corba object query service and its evaluation. To appear in: *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999.

[RSHN97] Lukas Relly, Hans-J. Schek, Olof Henricsson, and Stephan Nebiker. Physical database design for raster images in CONCERT. In *5th International Symposium on Spatial Databases (SSD'97)*, Berlin, Germany, July 1997.

[RSS98] Lukas Relly, Heiko Schuldt, and Hans-J. Schek. Exporting database functionality — the CONCERT way. *Data Engineering Bulletin*, 21(3):43–51, September 1998.

[SPSW90] Hans-Jörg Schek, H.-B. Paul, M.H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):25–43, March 1990.

[Sto86] Michael Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of the International Conference on Data Engineering*, pages 262–269, Los Angeles, CA, February 1986. IEEE Computer Society Press.

[SW93] Hans-Jörg Schek and Andreas Wolf. From extensible databases to interoperability between multiple databases and GIS applications. In *Advances in Spatial Databases: Proceedings of the 3rd International Symposium on Large Spatial Databases*, volume 692 of *Lecture Notes in Computer Science*. Springer Verlag Berlin Heidelberg New York, June 1993.

[SZB+96] Avi Silberschatz, Stan Zdonik, et.al. Strategic directions in database systems – breaking out of the box. *ACM Computing Surveys*, 28(4):764–778, December 1996.

A Interface Definition of the Kernel Concepts

CONCEPT UNKNOWN	
COPY_SIZE	object, copy_flags
COPY	source, target, copy_flags
DELETE_AUX	object

CONCEPT SCALAR ISA UNKNOWN		
COMPARE	o1, o2	$\rightarrow \{-1, 0, 1\}$

CONCEPT RECORD ISA UNKNOWN		
SUB_OBJECT_SIZE	object, component	$\rightarrow \mathbb{N}$
SUB_OBJECT	object, component	$\rightarrow$ part

CONCEPT LIST ISA UNKNOWN		
OPEN	object	$\rightarrow$ cursor
FETCH	cursor	$\rightarrow$ element
CLOSE	cursor	

CONCEPT SPATIAL ISA UNKNOWN		
OVERLAPS	object1, object2	$\rightarrow$ SCALAR_HASH
SPLIT	object	$\rightarrow$ { object }
COMPOSE	{ object }	$\rightarrow$ object
APPROX	{ object }	$\rightarrow$ object

B Example Kernel Modules

```
// "StringType" and "DateType" are predefined as SCALAR and SPATIAL concepts
CREATE CONCEPT SatelliteImage AS
       SPATIAL
  WITH OVERLAPS := SI_Overlaps.so
       SPLIT    := SI_Split.so
       COMPOSE  := SI_Compose.so
       APPROX   := SI_Approx.so

CREATE CONCEPT SatelliteData AS
       RECORD ( StringType, DateType, SatelliteImage )
  WITH SUB_OBJECT_SIZE := SD_GetSubObjSize.so
       SUB_OBJECT      := SD_GetSubObj.so
```

C Example Global Schema

```
repository WeatherServer := WebPageWrapper.so
{  URL := "http://www.weathernet.com/..."  };

// object types "Image", "Coordinate" and "Time" are predefined
class SatelliteImage ( extent SatelliteImages )
{
  attribute string title;
  attribute Time   date;
  attribute Image  picture;

  Image   window  ( x, y, w, h : integer ) := SI_window.so;
  boolean is_over ( pos : Coordinate )     := SI_translate.so;
};
```

```
class WeatherMeasurement@WeatherServer ( extent Measurements )
{
  attribute Time       when;
  attribute Coordinate where;
  attribute int        rainfall;
  attribute int        airPressure;
  ...
};
```