ObjectGlobe: Ubiquitous Query Processing on the Internet*

R. Braumandl M. Keidl A. Kemper D. Kossmann A. Kreutz
S. Pröls S. Seltzsam K. Stocker

Universität Passau
Fakultät für Mathematik und Informatik
D-94030 Passau, Germany

⟨*last name*⟩`@db.fmi.uni-passau.de`
`http://www.db.fmi.uni-passau.de/`

Abstract

We present the design of ObjectGlobe, a distributed and open query processor. Today, data is published on the Internet via Web servers which have, if at all, very localized query processing capabilities. The goal of the ObjectGlobe project is to establish an open market place in which *data* and *query processing capabilities* can be distributed and used by any kind of Internet application. Furthermore, ObjectGlobe integrates *cycle providers* (i.e., machines) which carry out query processing operators. The overall picture is to make it possible to execute virtually any kind of query operator on any machine and any kind of data on the Internet. One of the main challenges in the design of such an open system is to ensure security. We discuss the ObjectGlobe security requirements, show how basic components such as the optimizer and runtime system need to be extended, and present the results of performance experiments that assess the additional cost for secure distributed query processing.

1 Introduction

The World Wide Web has made it very easy and cheap for people and organizations all over the world to exchange *data*. Today, virtually everybody can publish a document by generating `HTML` (or `XML`) and placing it on some Web server; likewise, it is more or less standard to make data stored in relational (or other) databases publicly available on the Web by establishing form-based interfaces and by using CGI scripts or Servlets. WWW clients can retrieve individual documents by a simple “click” and they can get specific information from a database (behind the Web server) by filling out a form. In other words, WWW clients today can easily execute “point queries” (i.e., given URL, return document) and they can execute queries that can be handled by a single database behind a Web server.

*This research is supported by the German National Research Foundation under contract DFG Ke 401/7-1

The goal of the ObjectGlobe project is twofold. First, we would like to create an infrastructure that makes it as easy to distribute *query processing capabilities* (i.e., query operators) as it is to publish data and documents on the Web today. Second, we would like to enable clients to execute complex queries which involve the execution of operators from multiple providers at different sites and the retrieval of data and documents from multiple data sources. In contrast to Applets, all query operators should be able to interact in a distributed query plan and it should be possible to move query operators to arbitrary sites, including sites which are *near* the data. The only requirement we make is that all query operators must be written in Java and conform to the secure interfaces of ObjectGlobe.

We believe that our ObjectGlobe system can help to develop new application scenarios and new ways in which people and organizations interact on the Internet. An organization, for instance, could outsource all or part of its data processing to specialized providers on the Internet. As another example, WWW clients can *query* the Web and carry out different operations on different data sources. Providers could charge for data and new query operators. A data provider (e.g., a car dealer) could also be interested in participating in ObjectGlobe in order to supply its product catalog for free. Furthermore, ObjectGlobe can serve as an experimental platform in order to test new distributed query processing techniques; a researcher could implement a new distributed join method and test its performance with real data using our demo installation in Passau and other sites.

In some sense, the ObjectGlobe system can be seen as a distributed query processor. ObjectGlobe has a lookup service (i.e., a meta-data repository) which registers all data sources, operators, and machines on which queries can be executed. The lookup service is used by the ObjectGlobe optimizer in order to discover relevant resources for a query. The optimizer generates a query evaluation plan with the goal to execute the query with as little cost as possible. This plan is then initiated and executed by the execution engine. The design of all of these components has been addressed in previous work. Jini, for example, has a related lookup service [Wal99], and projects like Mariposa [SAL+96] or Garlic [HKWY97] (to name just two) have recently studied wide-area distributed query processing. What makes the ObjectGlobe system special is its "brutal" openness that in principle allows to execute any kind of (Java) operation on any machine and on any kind of data. One particular issue that needs to be addressed in this kind of system is "security" and how to protect data (and other resources) from unauthorized access. Another challenge is to ensure scalability in the number of sites. In this paper, we will describe the approaches we have chosen to address these challenges and give some initial performance results obtained using our system. The development of techniques for "schema integration" in a distributed and heterogeneous environment is not the target of our work because this has been addressed in other work (e.g., [SL90]); we assume that all data is in a standard format (e.g., relational or XML) or wrapped [RS97]. Although, "selling" services is one of the main motiviations for the ObjectGlobe project, the system does not require a particular business model; many different business models can be implemented on top of ObjectGlobe.

The remainder of this paper is structured as follows: Section 2 gives an overview of the ObjectGlobe system. Sections 3 and 4 describe the basic components of the system. Section 5 contains the results of some initial performance experiments conducted with ObjectGlobe on the Internet. Section 6 discusses related work. Section 7 concludes this paper.

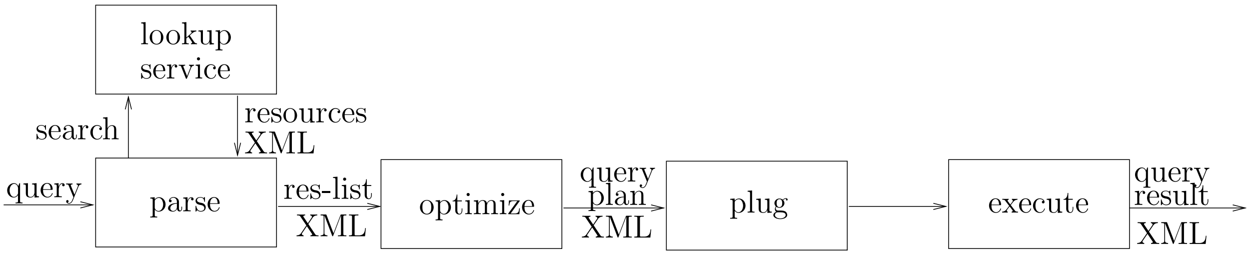


Figure 1: Processing a Query in ObjectGlobe

2 Overview of the ObjectGlobe System

The goal of the ObjectGlobe project is to distribute powerful query processing capabilities (including those found in traditional database systems) across the Internet. The idea is to create an open market place for three kinds of suppliers: *data providers* supply data, *function providers* offer query operators to process the data, and *cycle providers* are contracted to execute query operators. ObjectGlobe enables applications to execute complex queries which involve the execution of operators from multiple function providers at different sites (cycle providers) and the retrieval of data and documents from multiple data sources. In this section, we will outline how such queries are processed, give an example, and discuss the security requirements of the system.

2.1 Query Processing in ObjectGlobe

Processing a query in ObjectGlobe involves four major steps. These four steps are illustrated in Figure 1:

1. Lookup: In this phase, the ObjectGlobe lookup service is queried to find relevant data sources, cycle providers, and function providers that might be useful to execute the query. In addition, the lookup service provides the authorization data to determine what resources may be accessed by the user (or application) who initiates the query and what other restrictions apply for processing the query.

2. Optimize: From the information obtained from the lookup service, a cost-based query optimizer compiles a low-cost and valid (as far as user privileges are concerned) query execution plan. This plan is annotated with site information indicating on which cycle provider each operator is executed and from which function provider the external query operators involved in the plan are loaded.

3. Plug: The generated plan is distributed to the cycle providers and the external query operators are loaded and instantiated at each cycle provider. Furthermore, the communication paths (i.e., sockets) are established.

4. Execute: the plan is executed following an iterator model [Gra93]. In addition to the *external* query operators provided by function providers, ObjectGlobe has *built-in* query operators for selection, projection, join, union, nesting, unnesting, and sending and receiving data. If necessary, communication is encrypted and authenticated. Furthermore,

the execution of the plan is monitored in order to interfere and possibly halt the execution of the whole plan in case of failures.

The whole system is written in Java, for two reasons[1]. First, Java is *portable* so that ObjectGlobe can be installed with very little effort; in particular, cycle providers which need to install the ObjectGlobe core functionality can very easily *join* an ObjectGlobe system. The only requirement is that a site runs a Java virtual machine. Second, Java provides secure extensibility. Like ObjectGlobe itself, external query operators are written in Java, they are loaded on demand (from function providers), and they are executed at cycle providers in their own Java "sandbox" (more details in Section 4). To provide an external query operator a simple uniform interface must be implemented; in addition, the new external query operator must be registered in the lookup service. Likewise, data providers and cycle providers must register in the lookup service before they can participate.

ObjectGlobe supports a nested relational data model; this way, relational, object-relational, and `XML` data sources can very easily be integrated. Other data formats (e.g., `HTML`), however, can be integrated by the use of wrappers that transform the data into the required nested relational format; wrappers are treated by the system as external query operators. As shown in Figure 1, `XML` is used as a data exchange format between the individual ObjectGlobe components. Part of the ObjectGlobe philosophy is that the individual ObjectGlobe components can be used separately; `XML` is used so that the output of every component can easily be visualized and modified. For example, a user can browse through the lookup service in order to find interesting functions which he/she might want to use in the query. Furthermore, a user can look at and change the plan generated by the optimizer.

2.2 An Example

To illustrate query processing in ObjectGlobe, let us consider the example shown in Figure 2. In this example, there are two data providers, A and B, and one function provider. We assume that the data providers also operate as cycle providers so that the ObjectGlobe system is installed on the machines of A and B. Furthermore, the client (not shown in Figure 2) can act as a cycle provider in this example. Data provider A supplies two data collections, a relational table R and some other collection S which needs to be transformed (i.e., wrapped) for query processing. Data provider B has a (nested) relational table T. The function provider supplies two relevant query operators: a wrapper (*wrap_S*) to transform S into nested relational format and a compression algorithm (*thumbnail*) to apply on an image attribute of T.

Figure 3 shows a graphical representation with the most important annotations of the `XML` query plan for this example. (The real `XML` plan is given in Appendix A.) In particular, Figure 3 shows the *host*, *source*, and *codeBase* annotations. The *host* annotation of an operator indicates at which machine (i.e., cycle provider) the operator is executed; e.g., the *display* operator is executed at the client. The *source* annotation of a *scan* iterator indicates which collection is to be read. The *codeBase* annotation indicates from which function provider an external query operator is read. *scan*, *display*, and the *joins* are built-in operators so that they do not have a *codeBase* annotation.

[1]Currently, the optimizer is written in C++, but we are planning to rewrite it in Java.

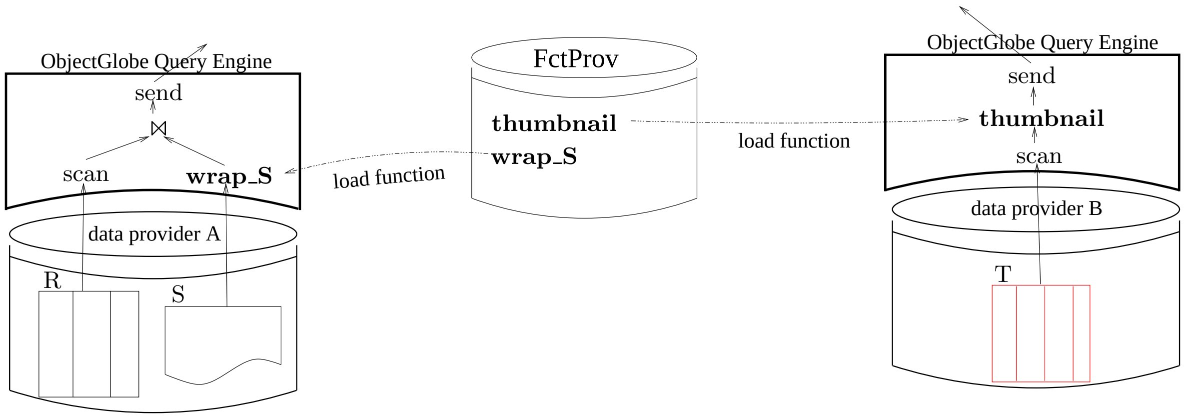


Figure 2: Distributed Query Processing with ObjectGlobe

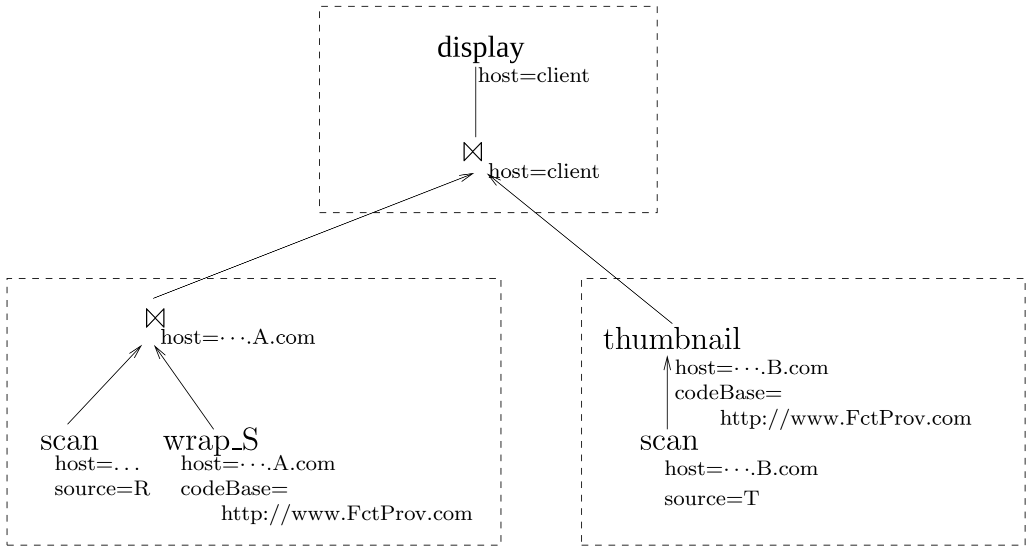


Figure 3: The Annotated Query Execution Plan

2.3 Security Requirements in ObjectGlobe

Safety is one of the crucial issues in an open and distributed system like ObjectGlobe. ObjectGlobe provides the infrastructure to deal with the following security issues:

Protection of Cycle and Data Providers: It has to be ensured that the resources of the cycle and data providers are protected from (possibly malicious) external operators loaded from unknown function providers. Based on the Java security model, all query operators are therefore executed in a protected area, a so-called *sandbox* (Section 4.3).

Secure Communication: The communication streams between the ObjectGlobe servers have to be protected against unauthorized access and modification. We adhere to the well-established standards (SSL and TLS) for encrypting and digitally signing messages between ObjectGlobe engines.

Authentication: ObjectGlobe supports a flexible authentication policy. Users and applications that only access free and publicly available resources can be anonymous and no authentication is required. If a user accesses a resource that charges and accepts electronic money, then the user can again stay anonymous and the electronic money is shipped as part of the "plug" step. Authentication is only required for authorization or accounting purposes of providers; in this case, users must provide certificates or other authentication information (e.g., passwords). Cycle providers can also require authenticated external operators to restrict the providers; e.g., to execute only code originating from trusted sources within the same company or Intranet.

Authorization: Some providers constrain the access or use of their resources to particular user groups. In general, providers apply their own autonomous authorization policy and control the execution of, say, query operators at their site themselves. In order to generate valid query execution plans and avoid failures at execution time, ObjectGlobe must know about these authorization constraints and register them in its lookup service.

3 Generating Query Plans

In this section, we show how ObjectGlobe produces a plan for a query; the next section then shows how such a plan is executed. Currently, ObjectGlobe supports a subset of SQL; ObjectGlobe, however, does support the use of external functions as part of a query. Like any other query processor, ObjectGlobe parses, optimizes, and executes a query. In this section, we describe the ObjectGlobe *lookup service* that finds relevant resources for a query and the parser and optimizer that try to find a good plan to execute a query.

3.1 Lookup Service

The lookup service plays the same role in ObjectGlobe as the *catalog* or *meta-data management* of a traditional query processor. Every provider must register its services before it can participate in ObjectGlobe. The ObjectGlobe parser and optimizer consult the lookup service in order to find all relevant resources to execute a query and get statistics. Furthermore, end users can use the lookup service to browse through the meta-data and search for available query capabilities and data sources for their applications. In some sense, therefore, ObjectGlobe's lookup service can also be seen as a *search engine* for *data* and *query processing capabilities*; it also bears some similarity with the Jini lookup service [Wal99] or with X.500 [CCI89] and LDAP directory services [WHK97]. The difference is that the ObjectGlobe lookup service is geared to the particular requirements of an open and secure distributed query processor.

3.1.1 ObjectGlobe Meta-data

The ObjectGlobe lookup service records the following information:

- **data provider:** each collection of objects stored by a data provider and the *attributes* of each collection are recorded by the lookup service. Each collection is associated to a *theme*; for example, `www.HotelBook.com` and `www.HotelGuide.com` provide two different collections associated to the theme *hotel.* A collection can be seen as a horizontal partition, but two collections of the same theme may have different attributes. In addition, all *iterators* that can be used to scan through a collection are recorded. Furthermore, the lookup service records if a collection provided by a data provider is a replica (i.e., mirror) of a collection provided by some other data provider.

- **cycle provider:** the CPU power, size of main memory, and temporary disk space of each cycle provider is recorded.

- **function provider:** the name and signature of each query operator is recorded. Furthermore, the requirements in terms of CPU speed, size of main memory and disk space to execute each query operator is kept by the lookup service. ObjectGlobe differentiates between *iterators* like join or display and *transformers* such as *thumbnail.* (In addition, ObjectGlobe also has special categories for *predicates* and *aggregate functions.*) Any kind of function, however, will automatically be wrapped by ObjectGlobe into an iterator so that we ignore these distinctions in this paper and use the words *function* and *query operator* interchangeably for the general concept.

- **statistics:** the lookup service stores any available information that helps the optimizer to estimate the cost (in $ and in response time) of a plan; e.g., histograms to estimate the selectivity of simple (i.e., non-external) predicates, latency and bandwidth of the interconnects between two cycle providers, typical load of cycle providers as a function of time, and possibly the `URL` of functions that can be used to estimate the cost of query operators.

- **authorization information:** the lookup service maintains *access control lists* that store which data may be processed at which cycle provider and by which query operator. It is also possible to restrict the execution of specific query operators at certain cycle providers. Following the ObjectGlobe authorization model, it is possible to specify positive and negative authorizations [RBKW91]. Also, it is possible to group collections, functions, and cycle providers into "authorization classes" in order to reduce the overhead of maintaining and processing this information in the lookup service.

To give a concrete example, Appendix B shows an example `RDF` document that can be used by a data provider that registers a *hotel* collection. It is important to keep in mind that all providers are autonomous and have their own local authorization policies. The meta-data kept in the lookup service mirrors that information and, thus, this meta-data can be outdated or incomplete. It is possible, for instance, that a data provider revokes the grant of some cycle providers to process its data without notifying the lookup service; as a result, the execution of an ObjectGlobe query might fail due to an authorization violation at execution time. ObjectGlobe relies on data, function, and cycle providers notifying the lookup service if important meta-data changes. If a plan fails due to stale meta-data in the lookup service, all the relevant meta-data is invalidated so that providers that do not update their meta-data do not participate in the ObjectGlobe federation. As an alternative, [CZH+99] propose to use a *time-to-live* scheme; in that scheme, providers must periodically contact the lookup service if they want to participate in the federation.

```
search DataProvider d
select d.uniqueId, d.attr.*
where d.theme.name = "hotel"
  and d.attr.?.topic = "city"
  and d.attr.?.topic = "price"
```

Figure 4: Example Search Query

```
<collection>
  <uniqueId> 4711 </uniqueId>
  <attr topic = "city", type = "String"/>
  <attr topic = "price", type = "US-Dollar"/>
  <attr topic = "address", type = "String"/>
</collection>
```

Figure 5: Example Search Result

3.1.2 Using the ObjectGlobe Lookup Service

As mentioned before, data, function, and cycle providers generate RDF documents in order to register their services. We use RDF because it is very flexible and a WWW standard for describing resources [BG99]. Typical providers, such as relational or XML data sources, can very easily be described using RDF; it is also possible to automatically produce large fractions of an RDF description from, say, an XML DTD or a relational schema. RDF is also used to update the meta-data if a provider changes or extends its services and the *id* of an RDF object is used to unregister (i.e., delete) services.

To find relevant resources and retrieve statistics and authorization information, the lookup service provides a declarative query language. As an example, Figure 4 shows how to ask the lookup service for all data providers that supply *hotel* collections. More specifically, the query of Figure 4 asks for *hotel* collections which have *city* and *price* attributes and the query asks for the *id* of the collection and information about all *attributes*. The result of this query is shown in Figure 5; here, we show the results for the *hotel* collection specified in the RDF document of Appendix B.

The lookup service also allows the definition of views. These views can be materialized. Such materialized views are very helpful to support sessions in which search results are iteratively refined. For example, it is possible to first ask for all cycle providers which are allowed to process objects of a specific collection and then, in a separate search request, ask which of *these* cycle providers are capable to execute a specific query operator.[2] This feature is important for parsing and optimization and for users who interactively browse the meta-data.

3.1.3 Implementation Details

The lookup service is currently implemented on top of a relational database system. Meta-data (i.e., RDF documents) are mapped to binary tables as described in [FK99]. Search requests are translated into SQL join queries. This translation is not one-to-one as the lookup service hides the details of how the meta-data is stored. Clients of the lookup service, for example, can ask for all cycle providers that are allowed to process objects of a specific collection; the lookup service will answer such a query considering all groups of cycle providers as well as all positive and negative authorizations. Translating search requests into SQL queries is quite complicated (albeit straightforward) and describing all the details is beyond the scope of this paper.

Currently, the lookup service is implemented as a centralized component of the ObjectGlobe system. Obviously, a centralized lookup service can easily become a bottleneck of the

[2] Of course, these cycle providers could also be found in a single search request.

whole system. Therefore, we are currently working towards a distributed/hierarchical lookup service (similar to the hierarchical name server we described in [EKK97]) with domain-specific replication and caching of meta-data.

3.2 Parser and Optimizer

Plans for a query are generated by the ObjectGlobe query parser and optimizer. As shown in Figure 1, the parser looks up the relevant resources for a query and the optimizer produces a plan based on (a subset of) these resources.

3.2.1 Parser

The main effort carried out during parsing is to issue search requests to the lookup service in order to discover all relevant resources (i.e., data, function, and cycle providers). The parser aborts the processing of a query if for some part of the query, no resources can be found. Not surprisingly, relevant data providers are found using the *themes* specified in the `FROM` clause of a query. For every *theme*, the lookup service registers no, one, or several *matching* collections from different data providers; a collection matches if it has the same theme and its objects have all the attributes used in the `SELECT` and `WHERE` clauses of the query. A typical search request to find relevant data providers is given in Figure 4. Likewise, the parser looks for function providers for each external function used in a query; again, external functions such as *thumbnail* can have several implementations from different function providers; all implementations that match the right name and signature are considered. Query operators such as *join, union,* or *display* are typically implicit in a query; for *join* and *union* the parser will consider built-in variants and all variants provided by function providers. For *display*, the parser will always consider ObjectGlobe's built-in variant which produces `XML` to represent query results; the parser will only consider a different *display* operator if this is explicitly requested.

In theory, every cycle provider can be useful to execute a query. Considering *all* cycle providers for every individual query would simply be infeasible. To find relevant and *interesting* cycle providers, data and function providers can register a set of *preferred cycle providers* to handle their data or execute their functions; this set of preferred cycle providers will typically include the machines of the data or function provider. In addition, each ObjectGlobe end user (or application programmer) can specify a set of preferred cycle providers; this set may include the client machine of the user. For a given query, the parser determines the overall set of interesting cycle providers from the preferred cycle providers of the user and of all relevant data and function providers. From this set, the parser will further prune cycle providers which are clearly not useful; e.g., cycle providers which are not allowed to process any function. It should be noted that registering preferred cycle providers is optional; therefore, it is possible that the parser stops processing a query if neither the user nor any relevant data or function provider have specified preferred cycle providers, although the query could be executed using *non-preferred* sites.

In addition to discovering all relevant resources, the parser consults the lookup service in order to retrieve all available statistics and authorization information. As a result, the parser produces a (quite complex) `XML` document which is then used by the optimizer in order to generate a plan. Figure 6 shows how the authorization and applicability information is

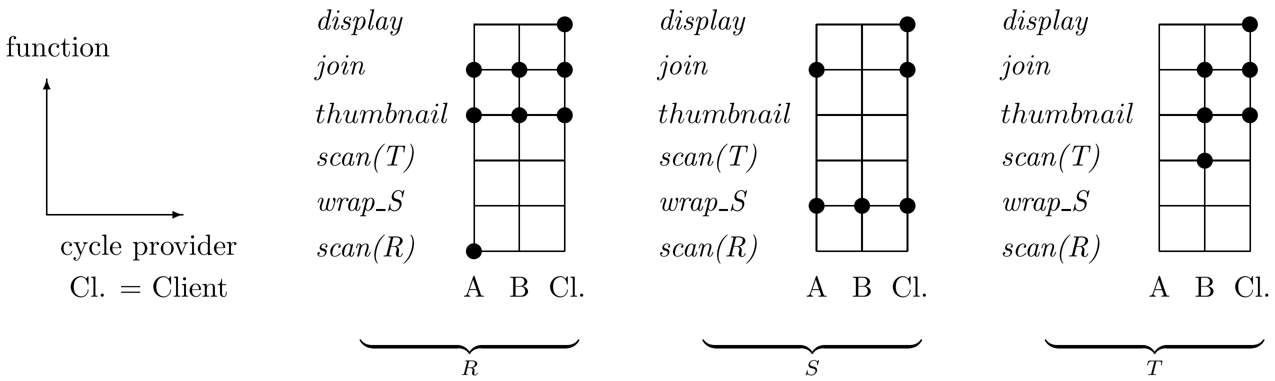


Figure 6: Compatibility Matrices for the Example of Section 2.2

represented as a *compatibility matrix* for the collections, functions, and cycle providers of the example of Section 2.2. For each relevant collection such a compatibility matrix is generated by the parser. A point at (c, f) in a matrix of a collection is set if cycle provider c is authorized to see the collection, function f is authorized to process objects of the collection, c is authorized to execute f, and c is capable of executing f (i.e., has enough memory and disk space). For instance, *wrap_S* may be executed at all cycle providers in order to read collection S, but it may obviously not be used anywhere to read collection R or T. In the matrix, built-in query operators such as *display, scan*, and *join* are treated in the same way as external functions (e.g., *thumbnail* and *wrap_S*); it would be possible, for instance, that a cycle provider only allows its own join methods to be executed on its machines.

3.2.2 Optimizer

The goal of the optimizer is to find a good plan to execute a query, if a plan exists. The "if a plan exists" part is important because the ObjectGlobe optimizer, other than a traditional optimizer, might sometimes fail to find a plan, even if the parser was able to discover relevant resources. First of all, limitations due to authorizations can make it impossible to find a valid plan; for instance, it might happen that two collections cannot be joined because there is no cycle provider that has permission to see both collections. Furthermore, ObjectGlobe users and applications can limit the total cost (in $) and response time of a query; this is an idea adopted from Mariposa [SAL+96].

The optimizer enumerates alternative plans using a System-R style dynamic programming algorithm. That is, the optimizer builds plans in a bottom-up way: first so-called *access plans* are constructed that specify how each collection is read (i.e., at which cycle provider and with which *scan* or *wrapper* operator). After that, *join plans* are constructed from these *access plans* and (later) from simpler *join plans.* To deal with unary external functions and predicates, the dynamic programming algorithm is extended as described in [CS96]. In every step, the cost of each plan is estimated and inferior plans are pruned in order to speed up the optimization process. Rather than presenting the full details of the ObjectGlobe optimizer, we would like to highlight the peculiarities that make the ObjectGlobe optimizer special:

Compatibility Matrix During query optimization every plan is annotated (among others) with a compatibility matrix. The compatibility matrix of an access plan is identical with the compatibility matrix generated by the parser for the corresponding collection. The matrix of a join plan which is composed of two sub-plans is generated by ANDing the two compatibility matrices of the two sub-plans, resulting in a more restrictive matrix.

Sanity Checks Some sub-plans can be immediately discarded during plan enumeration based on the sub-plan's compatibility matrix. As an example, consider the following situation: collections R_1 and R_2 belong to the same theme $\mathcal{R}$ and a query is interested in $f(\mathcal{R})$ for some external function f. For collection R_1, f may only be executed by cycle provider x; for collection R_2, f may only be executed by cycle provider y. Now a sub-plan $R_1 \cup R_2$ can immediately be discarded because there is no way to execute f on top of $R_1 \cup R_2$ (neither x nor y work); in other words, the $R_1 \cup R_2$ plan has no points set in the f row of its compatibility matrix. (Note, however, that an $f(R_1) \cup f(R_2)$ plan is valid, if it is equivalent.) If several variants of f exist, then the $R_1 \cup R_2$ plan can be discarded if there is no point set in the *shelf* of f rows. (A shelf is a set of rows in the matrix for different variants of the same function.) Obviously, a plan can also be immediately discarded if its estimated cost or response time exceeds the specified limit.

We also carry out more sophisticated sanity checks at the beginning of query optimization. For example, there must be at least one cycle provider which has permission and is capable to execute the *display* operator for each collection. Typically, this must be the client machine at which the query was issued. If such a cycle provider does not exist, then no plan exists and the optimizer can stop without enumerating any plans. In theory, such sanity checks that span several compatibility matrices could be applied in order to discard certain sub-plans during the plan enumeration process; since these sanity checks are quite costly, however, they are only carried out once, at the beginning before plan enumeration starts.

UNION Queries As shown earlier, collections can be horizontal partitions which need to be *unioned* and different collections of the same theme can have different authorization requirements (i.e., different compatibility matrices). As a result, the optimizer must consider each collection individually, even collections of the same theme which are not treated individually by traditional optimizers. Considering every collection individually involves extending the dynamic programming algorithm for plan enumeration; essentially, the optimizer enumerates $R_1 \cup R_2$ in the same way as a two-way join plan and $R_1 \cup R_2 \cup R_3$ in the same way as a three-way join plan, if R_1, R_2, R_3 belong to the same theme. The ObjectGlobe optimizer would also consider plans like $(R_1 \cup R_2) \bowtie S$ as well as $(R_1 \bowtie S) \cup (R_2 \bowtie S)$ for queries that involve these three collections.

Bypassing Data Currently, the optimizer models authorization information in a coarse grained way: either the whole collection or nothing may be processed by a cycle provider or some function. In many application environments, however, a finer-grained authorization model is desirable; e.g., a cycle provider might be allowed to see certain attributes of a collection and others not. Such a finer-grained authorization model makes more plans possible. One option is to encrypt those attributes that a cycle provider is not allowed to see. Another option is to *split* the objects of a collection and only ship those attributes to the cycle provider which the cycle provider is allowed to see; the other attributes are bypassed and later the objects are reassembled at a cycle provider that has permission to see all attributes. In some sense, such a plan works like a semi-join program. In order to enumerate such plans, we have once more extended the plan enumeration algorithm; details of this extension are described in [BKKS99]. (As shown in [BKKS99], such plans are also useful to get better response times even if there are no authorization constraints.)

Costing To estimate the cost (in $) and response time of a plan the optimizer completely relies on the statistics and cost functions registered in the lookup service. In the absence of such statistics, the ObjectGlobe optimizer will *guess*, just as any other optimizer. More work on costing in distributed and heterogeneous query processors has been reported in [ROH99]. Costing in the ObjectGlobe optimizer is currently very simple, but we are planning to extend our costing framework along the lines proposed in [ROH99].

IDP Evidently, the search space can become too large for full dynamic programming to work for complex ObjectGlobe queries. To deal with such queries, we developed another extension that we call *iterative dynamic programming* (`IDP` for short). `IDP` is adaptive; it starts like dynamic programming and if the query is simple enough, then `IDP` behaves exactly like dynamic programming. If the query turns out to be too complex, then `IDP` applies heuristics in order to find an acceptable plan. Details and a complete analysis of `IDP` is given in [KS98b].

4 Query Plan Distribution and Execution

As mentioned before, ObjectGlobe was implemented in Java for two reasons: portability and security. In this section we will describe how we utilized Java's features to achieve extensibility and query operator mobility without compromising security. We will also describe ObjectGlobe's monitor concept for controlling the progress of distributed query plans.

4.1 Distributing Query Evaluation Plans

Query plans are distributed in a straightforward way using the *host* annotations of the iterators in the plan. Every cycle provider loads the code of the external operators with a specialized ObjectGlobe class loader (OGClassLoader); the `URL` of the code is given in the *codeBase* annotation. If a cycle provider requires that the code is signed (authenticated), then the OGClassLoader will check the signature of the code. Furthermore, all communication paths are established by (built-in) send and receive iterators. If desired (i.e., specified in the annotations of the plan), an SSL (Secure Sockets Layer) connection is established. SSL [FKK96][3] is a defacto standard for providing privacy and reliability of network communication by encrypting network traffic and checking the data integrity using Message Authentication Codes (MAC). Also, the SSL protocol can carry out the authentication of both ObjectGlobe communication partners via certificates. Finally, authorization is carried out by the individual providers when

[3] There is also the standardized TLS (Transport Layer Security) protocol [DA99] of the Internet Engineering Task Force (IETF) which is quite similar to the current SSL 3.0 protocol.

a query is instantiated. If a provider restricts the use of its resources, the authentication information of the user will be part of the plan (again, as part of an annotation). Two possible authentication schemas are supported. Parts of a plan can be signed and carry the corresponding certificate. Another way of authenticating clients, which is still in use by many providers, is password-based. In this case the corresponding sub-plan is annotated with the encrypted password.

4.2 Extensibility

To integrate an external function, a function provider must implement a simple predefined interface. To implement an *iterator*, for example, `open`, `next`, `close`, and `reopen` methods must be implemented following the iterator model described in [Gra93]. The interface of other external functions (e.g., *transformers* such as thumbnail) is simpler; these external functions are wrapped by generic (built-in) ObjectGlobe iterators.

In the following we briefly describe the `open` method for iterators, since it has a special requirement. The `open` method returns an object of a class named `TypeSpec`. Such an object describes the type of the tuples which will be produced with every call of the `next` method. Type specifications are also recorded in the lookup service; just like authorization information, however, the type specifications recorded in the lookup service might be outdated or incomplete. Based on these (runtime) `TypeSpecs` polymorphic functions can be constructed. Furthermore, it is possible to compute the *union* of two collections that have different attributes; for example, two *hotel* data sources on the Internet (e.g., `www.HotelBook.com` and `www.HotelGuide.com`) might have slightly different attributes and it is nevertheless possible in ObjectGlobe to ask a `select *` query that retrieves all attributes from both sources.

4.3 Secure Query Engine Extensibility

We have utilized Java's security model [Oak98] to guarantee security of ObjectGlobe servers while executing external operators from possibly unknown function providers. Java's five-layer security model is illustrated in Figure 7. Java is a strongly typed object-oriented programming language with information hiding. The adherence to typing and information hiding rules are verified by the compiler and again by the class/bytecode-verifier before a Class object is generated from the bytecode because code could be generated by an evil compiler. The class loader's task is to load the bytecode of a class into memory, monitor the loaded code's origin (i.e., its `URL`) and to verify the signature of the authenticated code. The security manager controls the access to safety critical system resources such as the file system, network sockets, peripherals, etc. The security manager is used to create a so-called *sandbox* in which untrusted code is executed. A special, particularly restrictive sandbox is used, for example, by Web browsers to execute Applets. The ObjectGlobe system is based on the latest Java Release 2, in which the Security Manager interfaces with the Access Controller. The Access Controller verifies whether an access to a safety-critical resource is legitimate based on a configurable policy, which is stored in the `PolicyFile`. Privileges can be granted based on the origin of the code and whether or not it is digitally signed (i.e., authenticated) code. In addition, the Access Controller allows to temporarily give classes the ability to perform an action on behalf of a class that might not normally have that ability by marking code as *privileged*. This feature is essential, e.g.,

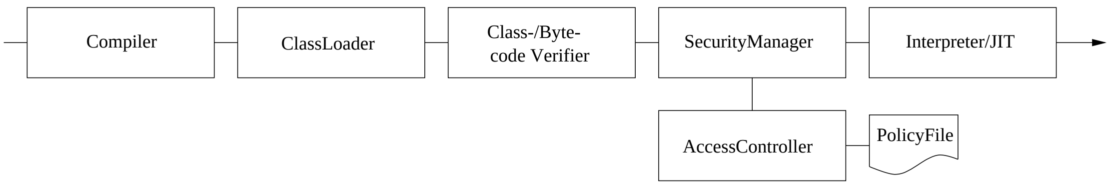


Figure 7: Java's Five-Layer Security Model

for granting access to temporary files as explained below. Finally, the Java program is executed by the interpreter (the JVM) which is responsible for runtime enforcement of security by checking array bounds and object casts, among others. From a security perspective, it is irrelevant whether or not parts of the code are compiled by a just-in-time (JIT) compiler to increase performance.

Of course, it would be unreasonable to grant unprotected access to system resources—such as the file system, the network sockets, etc—to unknown code. Therefore, all external operators are executed in a "tight" sandbox. Furthermore, the name spaces of concurrent queries are separated from each other. This way it is guaranteed, that they cannot illegitimately exchange information via "hidden communication paths", e.g., via static class variables of external operators. The name space separation is achieved by using a new, dedicated class loader (called *OGClassLoader*) for each query. This class loader is responsible for loading any additional functions beyond the built-in ObjectGlobe classes. The code bases (i.e., the function providers) from which these operators can be loaded are annotated in the query execution plan. Schematically, the name space separation and the class loaders are illustrated in Figure 8.

Some query operators require access to the cycle provider's secondary memory in order to store temporary results. Obviously, we cannot generally grant access to the file system to any external operator. Instead, a particular built-in class, called *TmpFile* has to be used. This built-in class provides a safe interface to create a temporary file, to write into and read from the temporary file and to delete the temporary file. Furthermore, a *TmpFile* object ensures the automatic deletion of the corresponding file when it is garbage collected. This way it is guaranteed that external operators can only operate on temporary files that they created themselves (within the same query execution plan). This scenario is illustrated in Figure 9.

Access to network sockets is normally prohibited to external operators to prevent them from sending any information about the data they process (to unknown locations). This restriction needs to be relaxed when a cycle provider wants to execute a wrapper which accesses data that is published by, e.g., a Web server. Therefore the policy of the Access Controller must be configured to allow a trusted and authenticated wrapper to establish a connection to a particular host on a given port. It is also possible to configure a relaxed policy that gives this privilege to arbitrary wrappers. The more restrictive policy situation is, for example, suitable for a wrapper accessing an `FTP` server to fetch a file. Granting the right to connect to this server to any external operator would allow operators to store any kind of information at this server, which is certainly not desirable. The more relaxed policy is applicable if granting access to a server is harmless; e.g., access to a server which only sends up-to-date exchange rates for given currencies.

The sandbox security model cannot protect providers from so-called denial of service attacks

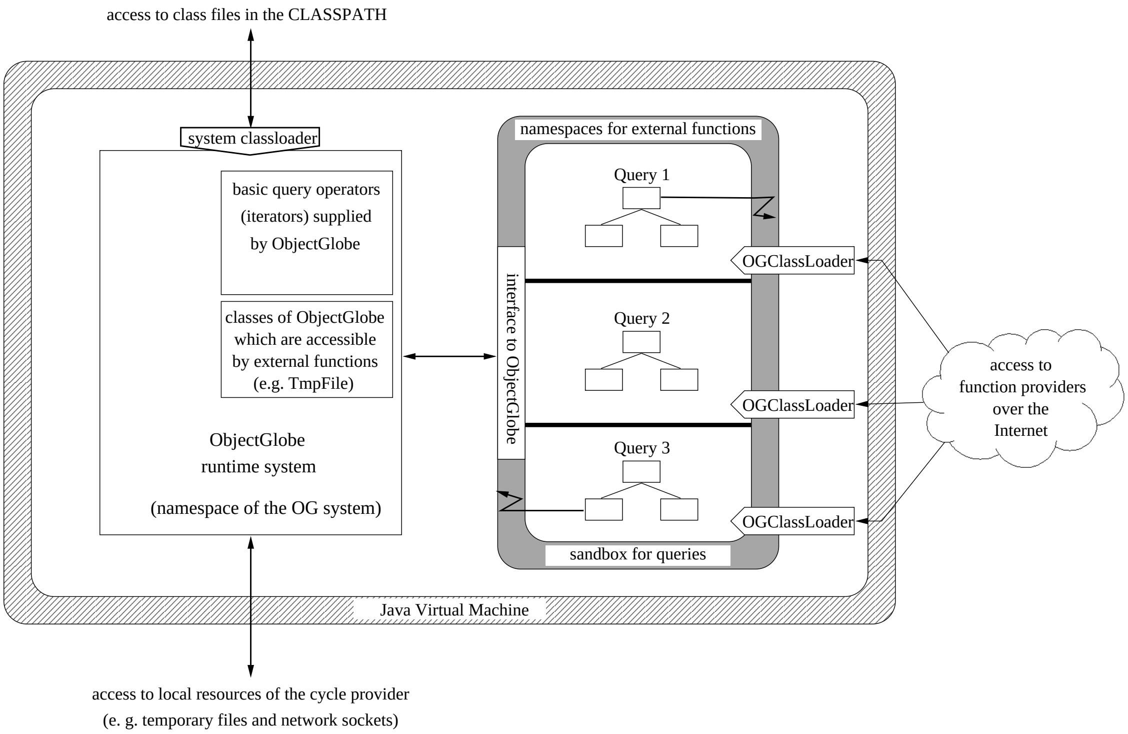


Figure 8: Security of Dynamically Loaded Code

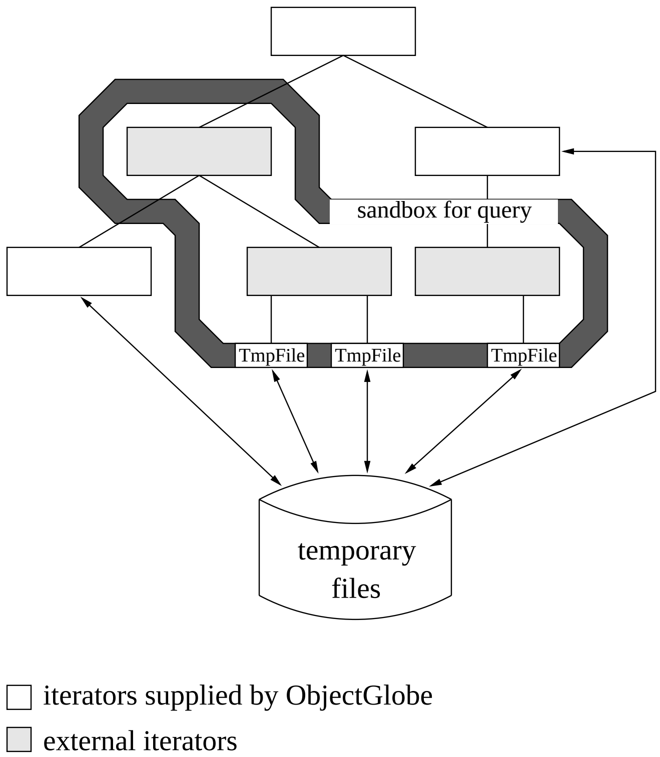


Figure 9: Extending Privileged Access Rights to External Operators

where malicious code overconsumes CPU cycles or other resources. To protect cycle or data providers from this kind of attack, accounting and authentication can help for identifying and possibly punishing intruders. As a part of a general accounting mechanism we will describe our monitor component which is used to control the progress of query operators. This way some simple overconsumption problems, such as operators which maliciously or accidentally consume resources without producing results, can be detected and repaired by halting the query execution.

4.4 Monitoring the Progress of Query Execution

The execution of an ObjectGlobe query can fail for a variety of reasons: network failures, crashed servers, badly programmed external operators, extremely overloaded servers, etc. Without precautions such failures can lead to live- or deadlocked query execution plans, in which upper-level query operators wait indefinitely for blocked sub-plans to deliver their results. Therefore, it is important to monitor the progress of the query execution and inform the participating ObjectGlobe servers about failures.

Each ObjectGlobe server uses a dedicated thread (we call it the *monitor thread*) for detecting blocked queries. A monitor thread operates on a data structure, which is organized as a priority queue. The objects stored in this queue represent future points in time and the object with the closest point in time has the highest priority. Such an object (we call it a *timeout object*) specifies an event inside a query, which has to occur in that query until the specified point in time has been reached. If its time has come, the monitor thread removes the timeout object out of the queue and checks if the associated event has occurred. If this is the case, the object is discarded and nothing else happens. Otherwise the affected sub-plan of the query is assumed to be blocked and it is terminated by a special terminator thread. When a sub-plan is stopped due to an error condition in an operator, the ObjectGlobe servers, executing the operators beneath and above the failed one in the plan hierarchy will be informed about this fact. The sub-plans of the operators beneath will normally fail. The operators above could react to the failure in special ways (also fail, rearrange the plan, execute an alternative sub-plan, etc. [CD99]). The propagation of an error up the hierarchy is performed by the standard exception handling mechanism of Java with a little help from our send-/receive operator pair for crossing network connections. The servers of child operators cannot be informed with the exception mechanism. A special (UDP) network protocol is used for this purpose.

What we did not mention up to now is, where the timeout objects come from. These objects are created by a special type of operator, a *monitor operator*. A monitor operator can be inserted at arbitrary positions in a query evaluation plan, since it does not change its input tuple stream. Positions where we will always insert monitor operators are above receive operators and above any external operator. Its job is to observe the progress of the actions performed by the sub-plan beneath it. For example, at the beginning of its open method a monitor operator creates a timeout object for the event "end of open reached" and inserts this object into the priority queue of the monitor thread, while also keeping a reference to that object. After that, the open method of its child operator is called. When the method invocation returns, the timeout object is informed, that its awaited event has occurred.

The advantage of this architecture is that the decisions about where to monitor in a query evaluation plan and with what parameters the timeouts should be initialized can be made in a

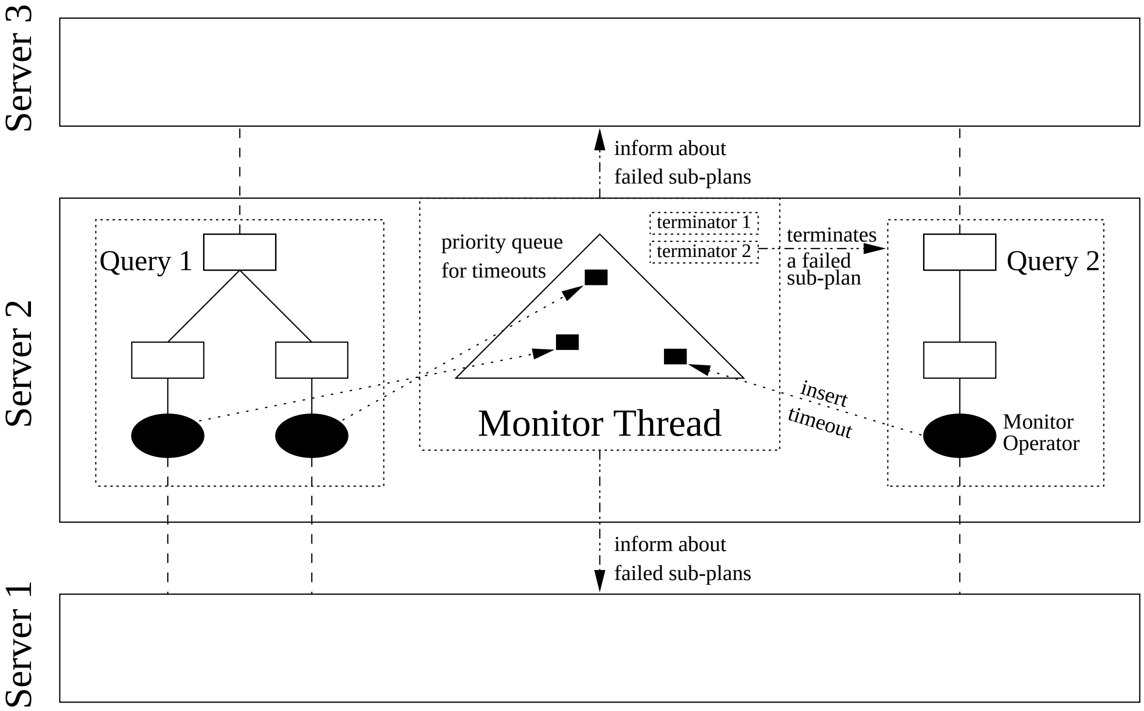


Figure 10: The Architecture of the Monitor Component.

flexible manner. Setting timeouts is critical, just as in any other system. One option is to set the timeout based on the response time estimates of the optimizer. Another option is to use a default value. Other operators and especially external operators need not implement anything for the monitor component. An overview of this architecture is given in Figure 10.

5 Performance Experiments

5.1 Overheads of Plan Generation

To determine the overheads of plan generation, we measured the *lookup* and *optimize* steps of processing a five-way join query[4]. The optimizer ran on a Sun Ultra 10 workstation; the lookup service ran on a Sun Ultra 1 workstation. There were six relevant cycle providers and the optimizer considered three different join variants (nested-loops, hash, and sort-merge). We studied two different scenarios. In Scenario I, all joins could be executed at all cycle providers; in Scenario II, joins with two of the five collections could only be executed at one specific cycle provider. Table 1 summarizes the results. Even though the meta-database of the lookup service is very small, most of the time is consumed in the lookup step; the reason is that twelve search requests are required for this query and the overhead of each search request is very high; clearly, we need to tune this in future work. The optimization time is acceptable in this experiment (< 1 sec). The optimization time is much lower for Scenario II than for Scenario I because the search space is much smaller for Scenario II due to the authorization restrictions.

[4]The optimizer is written in C++; everything else in Java.

	Total Lookup Time	Avg. Time per Search	Optimization Time
Scenario I	5.64 secs	0.47 secs	0.83 secs
Scenario II	5.64 secs	0.47 secs	0.07 secs

Table 1: Overheads of Plan Generation

5.2 Query Execution Times

5.2.1 Benefits of Operator Mobility

The following experiment illustrates the benefits of ObjectGlobe which makes it possible to execute query operators near data sources. We measured the execution time of a query which requests information about all hotels located in Philadelphia from HotelBook, an Internet data source. To perform this task a wrapper was used which first queries a list of all hotels in a given city and afterwards queries detailed information for every single hotel in this list one at a time, according to the query capabilities of the data source. There are two ways to execute this query. The traditional one is to execute the wrapper at the local machine in Passau, the other one which is made available by ObjectGlobe is to execute the wrapper at a host near the data source. Because it is impossible to execute the wrapper at the host serving HotelBook in Phoenix, we used a host in Maryland for this experiment.

As the results in Table 2 show there is a clear benefit if the wrapper is executed in Maryland because Maryland is closer to the HotelBook database than Passau and therefore the latency time is reduced when the wrapper iteratively accesses the HotelBook database. This experiment does not demonstrate how parallelism can be used to speed up query execution, but performance gains from parallelism can also be achieved with ObjectGlobe.

wrapper execution location	time
Passau	151 secs
Maryland	62 secs

Table 2: Query for all Hotels in Philadelphia

5.2.2 Costs of Secure Communication

SSL sockets [FKK96] and therewith encryption and Message Authentication Codes (MACs) are an effective way to integrate secure communication into a distributed system. But cryptographic algorithms have additional costs when transmitting data across a network. To demonstrate this effect we executed a simple scan-display plan and varied sites of the scan operator and the usage of SSL. In all cases the scan operator had to process 10 MB of data. As Table 3 illustrates, costs for encryption and MAC calculation can be neglected in a WAN environment. The first column contains information about where the scan and the display operators were executed[5] and across what kind of network the data was sent. The remaining three columns list the times of query

[5] $X \rightarrow Y$ means that the scan operator was executed at host X and the display operator was executed on host Y.

executions where the data was not encrypted and no MAC was calculated (plain), where only a MAC was calculated (SHA) and where both, encryption and MAC calculation, were done (IDEA + SHA). Whereas the first row shows that secure communication increases the query execution time in LAN environments (but the overall execution time is even with fully secured communication neglectable compared to query executions in a WAN environment) the second row shows that in a WAN environment there is no significant time difference between secure and insecure query execution because costs for cryptographic algorithms are CPU costs and are superimposed by communication costs.

	plain	SHA	IDEA + SHA
scan [Passau → Passau], 100 MBit LAN	3.54 secs	5.31 secs	11.86 secs
scan [Mannheim → Passau], WAN	81.93 secs	81.86 secs	82.04 secs

Table 3: Costs of Secure Communication in Different Network Environments

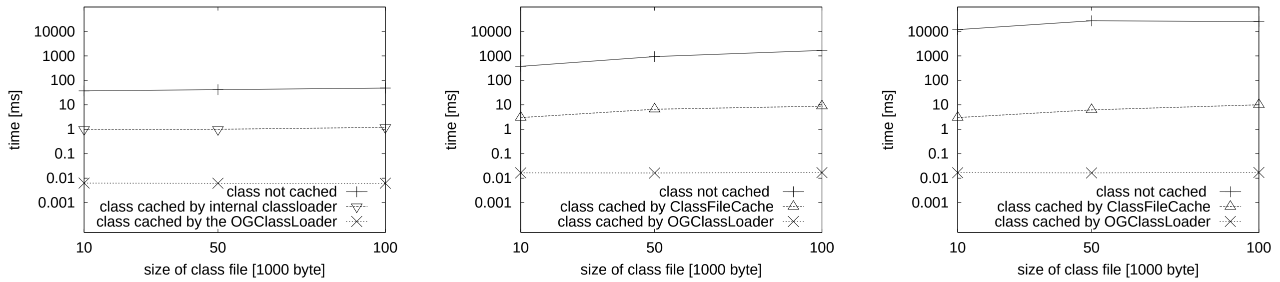
5.2.3 Costs of Dynamic Extensibility

One of the prominent features of ObjectGlobe is its dynamic extensibility by external operators. There are of course additional costs caused by loading classes from the network and the separation of name spaces of different queries compared to loading locally available built-in operators. This separation of name spaces is achieved by using an individual OGClassLoader for every query and it forbids the caching of Class objects for external operators. We only can cache the bytecode of external operators in the ClassFileCache. To measure these costs and how they can be reduced using caches we loaded classes of different size from different locations: built-in operators from disk, operators from a local function provider and classes from a remote function provider in Maryland. Figure 3 shows the following results:

- The costs for the initial loading of a class from disk or network are very high but can be heavily reduced by caching the class object (in the case of built-in operators) or caching the bytecode (in the case of external operators).
- Apart from the initial loading, the only costs for loading classes in a system without security would be the costs for fetching classes from the cache of the OGClassLoader (given by the X-lines in the graphs). In a system with security the typical costs for loading classes are shown by the lines marked with triangles. As expected, the costs for security are relatively high but compared to the total costs of query execution the costs for security can be neglected.

6 Related Work

In the design of ObjectGlobe we were influenced by work in two fields: (1) distributed and extensible database systems and (2) open distributed object systems. Distributed database systems have been studied since the late seventies in projects like System R*, SDD-1, or Distributed



a) Built-In Operator b) Ext. Op./Local Prov. c) Ext. Op./Remote Prov.

Figure 11: Costs of Loading an Operator by the ObjectGlobe Class Loader

Ingres. ObjectGlobe shares with all these projects the vision that a distributed database system can be used as easily as a centralized system (i.e., transparency) and that good performance can be achieved by sophisticated query optimization. The notion of an open market place in which different providers compete for queries is borrowed from Mariposa [SAL+96]—even though, ObjectGlobe does not enforce a particular business model like Mariposa. Extensibility has been studied in a number of database projects; e.g., Postgres [SR86], Starburst [HCL+90], or more recently in Predator [SLR97]. The safe execution of external functions has been studied in [GMSvE98], but the scope of that work is too limited for our context. Of course, we have also learned from recent projects in the design of query processors for distributed and heterogeneous data (so-called *mediators*); e.g., Garlic [C+95], TSIMMIS [PGGMU95], DISCO [TRV98], Tukwila [IFF+99], or the Information Manifold [LRO96]. These projects showed that it is possible to integrate any kind of data and exploit query capabilities of diverse data sources. Most heterogeneous database systems, however, rely on a heavy middleware architecture and strongly on data shipping which does not perform well in many situations (see, e.g., [FJK96]); exploiting the portability of Java we are able to do much better in ObjectGlobe. In summary, ObjectGlobe tries to put together the pieces developed in all that previous work.

There has also been a large body of related work on the integration of services in open distributed object systems. The most prominent examples are Jini [Wal99] and CORBA [MZ95]. A related lookup service is HP's Chai (Plug & Play) system [HPI99]. Architectures for distributed object systems have been devised in the SHORE [CDF+94], Ninja [GWBC99], and AutO [Kri98] projects. The AutO project was also conducted at the University of Passau and we adopted many results such as the AutO security model and infrastructure for ObjectGlobe. As part of the Ninja project, a secure distributed directory service has been developed [CZH+99]. What makes ObjectGlobe different from all these works is that ObjectGlobe is capable of complex query processing; that is, a single ObjectGlobe query can involve the lookup and execution of many different services and it requires optimization because of the large amounts of data that need to be processed.

In other lines of work, researchers have tried to "query the Web" using languages like WebSQL [MMM97, KS98a]; these efforts, however, only support a navigational style of access of Web pages. Junglee [GHR97] follows a data warehousing approach in order to integrate Internet data for query processing. Furthermore, Web site management has been studied in a few recent projects; e.g., Strudel [FFK+98]. The goal of systems like Strudel, however, is to improve the services (and manageability) of a single site, rather than integrating services from multiple sites.

7 Project Status and Future Work

We presented the design of ObjectGlobe, an open and distributed and secure query processing system. The goal of ObjectGlobe is to establish an open market place in which data, function, and cycle providers can *sell* their services, following some business model which can be implemented on top of ObjectGlobe. End users and applications can use these services in a low-overhead way. We gave details of the ObjectGlobe lookup service, the query parser and optimizer, and the runtime system. For each component, we showed the necessary adjustments in order to produce valid plans and guarantee security at execution time.

The project started about three years ago. A first demo was given at SIGMOD 99 [BKK99]. This demo involves two *hotel* servers (i.e., HotelGuide and HotelBook, located in the USA), a server with images of tourist attractions (located in Germany), a German server with *city* information, and the server of the German railways with all German train connections. This demo can be seen from the ObjectGlobe Web page. While some of the components by themselves are already quite sophisticated and highly tuned, our major efforts so far were in getting the system running. As future work, we are planning to improve the performance of the system. For example, we would like to reduce the number of queries that the lookup service needs to process during parsing. Furthermore, we would like to build data caches for ObjectGlobe. We also use the system to experiment with different query processing techniques and to find out what query processing on the Internet really costs.

References

[BG99] D. Brickley and R. V. Guha. Resource Description Framework (RDF) schema specification. Proposed Recommendation `http://www.w3.org/TR/PR-rdf-schema`, WWW-Consortium, March 1999.

[BKK99] R. Braumandl, A. Kemper, and D. Kossmann. Database patchwork on the internet (project demo description). In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 550–552, Philadelphia, PA, USA, June 1999.

[BKKS99] R. Braumandl, A. Kemper, D. Kossmann, and K. Stocker. Integrating semi-join reducers into state-of-the-art query processors. 1999. Submitted for publication.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, pages 124–131, March 1995.

[CCI89] CCITT. *Verzeichnis-Systeme*, volume 8 of *CCITT-Empfehlungen der V-Serie und der X-Serie*. R. v. Decker's Verlag, G. Schenk, Heidelberg, 1989.

[CD99] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Trans. Software Eng.*, 25(3):309–316, May 1999.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, Minneapolis, MI, USA, May 1994.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 87–98, Bombay, India, September 1996.

[CZH+99] S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proc. of ACM MOBICOM Conference*, Seattle, WA, August 1999.

[DA99] T. Dierks and C. Allen. The TLS Protcol Version 1.0. `ftp://ftp.isi.edu/in-notes/rfc2246.txt`, January 1999.

[EKK97] A. Eickler, A. Kemper, and D. Kossmann. Finding data in the neighborhood. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 336–345, Athens, Greece, August 1997.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 414–425, Seattle, WA, USA, June 1998.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 149–160, Montreal, Canada, June 1996.

[FK99] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. *IEEE Data Engeneering Bulletin*, 22(3):27–34, September 1999.

[FKK96] A. Frier, P. Karlton, and P. Kocher. *The SSL 3.0 Protocol.* Netscape Communications Corp., `http://home.netscape.com/eng/ssl3/ssl-toc.html`, November 1996.

[GHR97] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual data technology. *ACM SIGMOD Record*, 26(4):57–61, December 1997.

[GMSvE98] M. Godfrey, T. Mayr, P. Seshadri, and T. v. Eicken. Secure and portable database extensibility. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 390–401, Seattle, WA, USA, June 1998.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GWBC99] S. Gribble, M. Welsh, E. Brewer, and D. Culler. The MultiSpace: an evolutionary platform for infrastructural services. In *Proc. of the Usenix Annual Technical Conference*, Monterey, CA, June 1999.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HPI99] Hewlett Packard Inc. Chai: Internet business solutions. `http://www.chai.hp.com/`, 1999.

[IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution engine for data integration. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 299–310, Philadelphia, PA, USA, June 1999.

[Kri98] N. Krivokapić. *Control mechanisms in distributed object bases. synchronization, deadlock detection, migration*, volume 54 of *Dissertationen zu Datenbanken und Informationssystemen*. infix-Verlag, Ringstr. 32, 53757 Sankt Augustin, 1998. ISBN: 3-89601-454-4, Dissertation, Universität Passau, Germany.

[KS98a] D. Konopnicki and O. Shmueli. Information gathering in the world wide web: The W3QL query languge and the W3QS system. *ACM Trans. on Database Systems*, 23(4):369–410, Dec 1998.

[KS98b] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. December 1998. Submitted for publication.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[MMM97] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *Int. Journal on Digital Libraries*, 1(1):54–67, 1997.

[MZ95] T. J. Mowbray and R. Zahavi. *The Essential Corba – Systems Integration Using Distributed Objects*. John Wiley & Sons, Chichester, UK, 1995.

[Oak98] S. Oaks. *Java Security*. O'Reilly & Associates, Sebastopol, CA, USA, 1998.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query tranlation scheme for rapid implementation of wrappers. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 161–186, December 1995.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Trans. on Database Systems*, 16(1):88–131, March 1991.

[ROH99] M. Tork Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 599–610, Edinburgh, GB, September 1999.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, January 1996.

[SL90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonmous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 66–75, Athens, Greece, August 1997.

[SR86] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 340–355, Washington, USA, June 1986.

[TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling acccess to distributed heterogeneous data sources with DISCO. *IEEE Trans. Knowledge and Data Engineering*, 10(5):808–823, October 1998.

[Wal99] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). `ftp://ftp.isi.edu/in-notes/rfc2251.txt`, December 1997.

A The XML Representation of a Query Execution Plan

```
<?xml version="1.0" encoding='ISO-8859-1'?>

<iterator id="display" code="iterators.display" host="client">
  <iterator id="join1" code="iterators.NestedLoops"
            codeBase="http://www.FctProv.com/forGlobalUse"
            host="client">
    <predicate>Sb = Tb</predicate>

    <iterator id="join2" code="iterators.NestedLoops" host="***.A.com">
      <predicate>Ra = Sa</predicate>

      <iterator id="wrapperS" code="wrapper.wrap_S"
                codeBase="http://www.FctProv.com" host="***.A.com">
      </iterator>
      <iterator id="tbscanR" code="iterators.TbScan" host="***.A.com">
        <source>R</source>
      </iterator>
    </iterator>

    <iterator id="thumb1" code="thumbnail" codeBase="http://www.FctProv.com"
              host="***.B.com">
      <toThumbNail>picture</toThumbNail>

      <iterator id="tbscanT" code="iterators.TbScan" host="***.B.com">
        <source>T</source>
      </iterator>
    </iterator>
  </iterator>
</iterator>
```

B The RDF Registration Code for a Data Provider

In the sample RDF-description shown below, the relevant information about the data provider can be found enclosed in the `s:DataProvider` element. At the beginning of the data provider description we can find the URL, with which the data provider can be contacted, the theme (i.e., hotel) this data provider is associated with, a plain-text description of the content of the provider, etc. More interesting is the content of the `s:attributes` element. It contains the description of the type of the tuples, given by the data provider. In our case the type contains three attributes and for each attribute the name and the type of the attribute are specified. After that we can find some restrictions on the functions, which can be applied on the tuples of this data provider, and on the cycle providers, which may see the data of this data provider. At last we get a reference for the wrapper, which performs the necessary transformations to integrate the data provider into an ObjectGlobe system.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:s="http://www.db.fmi.uni-passau.de/objectglobe/schema.rdf#">
 <s:DataProvider rdf:ID="HotelBook">
     <s:url>http://www.hotelbook.com/cgi-bin/hsearch</s:url>
     <s:theme rdf:resource="http://www.db.fmi.uni-passau.de/objectglobe/common.rdf#hotel" />
     <s:content>Description of hotels worldwide.</s:content>
     <s:uniqueID>4711</s:uniqueID>
     <s:attributes><rdf:Bag>
         <rdf:li><s:Attribute>
           <s:topic rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/travel.rdf#city" />
           <s:domain rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/common.rdf#String" />
         </s:Attribute></rdf:li>
         <rdf:li><s:Attribute>
           <s:topic rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/travel.rdf#address" />
           <s:domain rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/common.rdf#String" />
         </s:Attribute></rdf:li>
         <rdf:li><s:Attribute>
           <s:topic rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/travel.rdf#price" />
           <s:domain rdf:resource=
              "http://www.db.fmi.uni-passau.de/objectglobe/common.rdf#US-Dollar" />
         </s:Attribute></rdf:li>
     </rdf:Bag></s:attributes>
     <s:allowedFP><rdf:Bag>
       <rdf:li rdf:resource="http://www.demo.de/leisure.rdf#hotelRanking" />
       <rdf:li rdf:resource="http://www.finance-demo.de/currency.rdf#currencyConverter" />
     </rdf:Bag></s:allowedFP>
     <s:allowAllFP>false</s:allowAllFP>
     <s:allowAllCP>true</s:allowAllCP>
     <s:wrapper
        rdf:resource=
          "http://www.db.fmi.uni-passau.de/objectglobe/wrappers.rdf#HotelBookWrapper" />
  </s:DataProvider>
</RDF>
```