ObjectGlobe: Ubiquitous Query Processing on the Internet*

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

Universität Passau, Lehrstuhl für Informatik, 94030 Passau, Germany; e-mail: `<lastname>@db.fmi.uni-passau.de`

Abstract We present the design of ObjectGlobe, a distributed and open query processor for Internet data sources. 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 a query with—in principle—unrelated query operators, cycle providers and data sources. Such an infrastructure can serve as enabling technology for scalable e-commerce applications, e.g., B2B and B2C market places, to be able to integrate data and data processing operations of a large number of participants. One of the main challenges in the design of such an open system is to ensure privacy and 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. Another challenge is quality of service management so that users can constrain the costs and running times of their queries.

Key words Distributed Query Processing – Query Optimization – Open Systems – Cycle-, Function- and Data Provider – Security – Privacy – Quality of Service

---

* This research is supported by the German National Research Foundation under contract DFG Ke 401/7-1 and the German Isreali Foundation (GIF).

** Current address: Technische Universität München, Institut für Informatik, 81667 München, Germany; e-mail: kossmann@in.tum.de

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.

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. Thus, distributed query plans can be composed of arbitrary query operators obtained from various function providers; 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 or a real estate broker) could also be interested in participating in ObjectGlobe in order to supply its product catalog for free. Open, distributed query processing, as in ObjectGlobe, is an essential enabling technology for scalable Internet applications, such as business-to-business (B2B) e-commerce systems like SAP's electronic marketplace "mySAP.com" [SAP99] which comprises hundreds of companies. One of the key challenges is to facilitate query processing over the various heterogeneous data sources in order to build integrated product catalogs, match product availability with demand forecasts, or perform price comparisons for procurement.

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. Every time a new provider joins or leaves an ObjectGlobe federation the corresponding meta-data is added to or removed from the respective meta-data repository. 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 in a way, which fulfills the user's quality constraints. This plan is then initiated and executed by the distributed execution engines (i.e., the ObjectGlobe servers). 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], Garlic [HKWY97] or Amos*II* [JR99] (to name just a few) have recently studied wide-area distributed query processing. What makes the ObjectGlobe system special is its "brutal" openness that allows to execute a query with—in principle—unrelated query operators, cycle providers and data sources. This transparent adhoc integration of operators and functions is a demanding task for query optimization which must take into account the logical and physical properties of these operations. 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 cycle and data providers. On behalf of the users, this means that they must be able to constrain the execution of their queries in such a system regarding monetary costs, execution time and the amount of data involved in the execution. In other words, quality of service management is necessary so that the behavior of the system becomes predictable and the investment of a user to execute a query is guaranteed.

In this paper, we will describe the approaches we have chosen to address all 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] or [JR99]). In particular, we do not report on work on ontologies [BCV99]. We assume that all data is in a standard format (e.g., relational or XML) or wrapped [RS97]. Furthermore, we assume that there is a meta-schema that can be used to describe all relevant properties of all services; for example, the ObjectGlobe meta-schema specifies that data sources are described among others by the set of *themes* (i.e., collections) they provide, a set of access methods or wrappers to read theses collections and statistics about value distribution and the cost to read these collections. How to (semi-) automatically extract all this information is beyond the scope of this paper. The emergence of XML, however, has initiated a number of standardization approaches for various businesses. For instance, global schemas for the real estate and financial sectors have been proposed in [Pet99] and [Gur00], respectively. Based on such standards, ObjectGlobe can very well be used in order to implement the remaining infrastructure (i.e., secure and reliable query processing).

Although "selling" services is one of the main motivations for our project, the system does not require a particular business model; many different business models can be implemented on top of ObjectGlobe. Devising specific business models for data processing on the Internet is also beyond the scope of this paper.

The remainder of this paper is structured as follows: Section 2 gives an overview of the ObjectGlobe system and compares it with other system architectures. Sections 3 and 4 describe the basic components of the system. Section 5 discusses security concerns in different scenarios and shows the advantages of implementing an electronic marketplace on top of an ObjectGlobe system. Section 6 contains the results of some initial performance experiments conducted with ObjectGlobe on the Internet. Section 7 concludes this paper.

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. Of course, a single site (even a single machine) can comprise all three services, i.e., act as data-, function-, and cycle-provider. In fact, we expect that most data and function providers will also act as cycle providers. 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

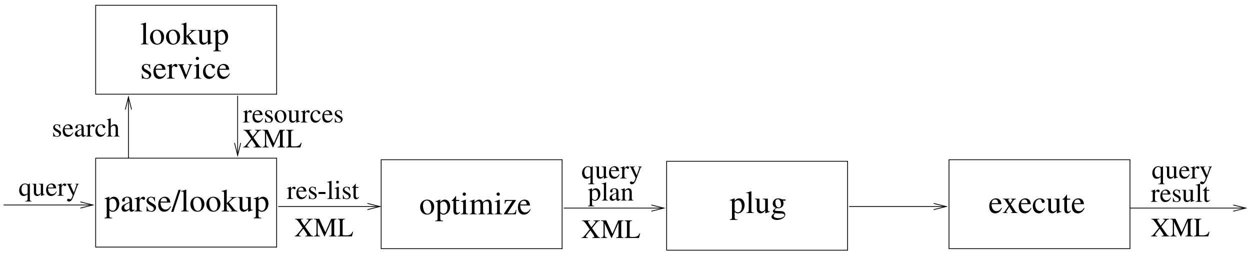


Fig. 1 Processing a Query in ObjectGlobe

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 (Figure 1):

1. Lookup: In this phase, the ObjectGlobe lookup service is queried to find relevant data sources, cycle providers, and query operators that might be useful to execute the query. In addition, the lookup service provides the authorization data—mirrored and integrated from the individual providers—to determine what resources may be accessed by the user who initiates the query and what other restrictions apply for processing the query.
2. Optimize: The information obtained from the lookup service, is used by a quality-aware query optimizer to compile a valid (as far as user privileges are concerned) query execution plan, which is believed to fulfill the users' quality constraints. 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 detect failures, look for alternatives, and possibly halt the execution of a plan.

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 the ObjectGlobe server on a Java virtual machine. Second, Java provides secure extensibility. Although many people complain about the execution speed of Java programs, we noticed that by avoiding some pitfalls in the Java I/O library the execution speed of the Java virtual machine is no bottleneck in wide area distributed systems. 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). Just like data and cycle providers, function providers and their external query operators must be registered in the lookup service before they can be used.

ObjectGlobe supports a nested relational data model; this way, relational, object-relational, and XML data sources can 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 be easily visualized and modified. For example, users can browse through the lookup service in order to find interesting functions which they might want to use in the query. Furthermore, a user can look at and change the plan generated by the optimizer.

2.2 Example Plans

To illustrate query processing in ObjectGlobe, let us consider the example shown in Figure 2—the corresponding query plan is sketched in Figure 3. The real XML plan is given in Appendix A. 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 can act as a cycle

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

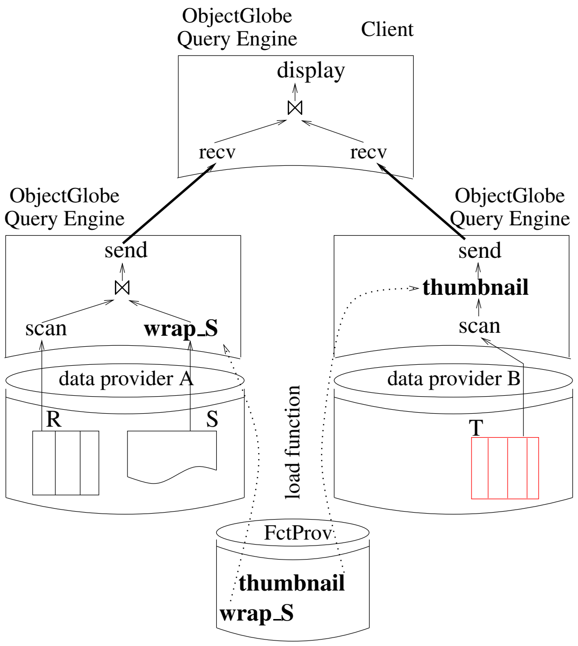


Fig. 2 Distributed Query Processing with ObjectGlobe

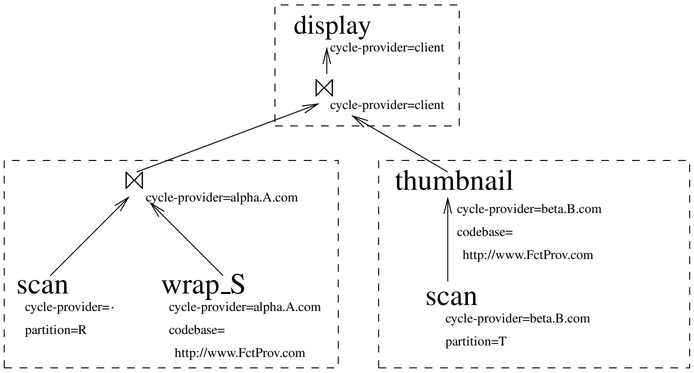


Fig. 3 Annotated Query Execution Plan

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 the most important annotations—in particular, the *cycle-provider*, *partition*, and *codebase* annotations—of the query plan. The *cycle-provider* annotation of an operator indicates at which machine the operator is executed; e.g., the final join and the *display* operators are executed at the client. The *partition* 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 loaded. *scan*, *display*, and the *joins* are built-in operators so that they do not have a *codebase* annotation.

Although the example above is rather small (in order to be illustrative) we expect ObjectGlobe systems to comprise a large number of cycle providers and far more data providers, with several of them contributing data to a specific theme. Figure 4 shows the structure of an example query which extracts information from a number of online databases that belong to different real estate brokers. The query uses a user-defined nearest neighbor operator (called *nn_10* in the figure) loaded from a function provider that is specialized on real estate data. The nearest neighbor logical operator is transitive and reflexive and hence allows us to perform the search for the ten nearest neighbors of a user-defined feature vector by first computing the ten nearest neighbors at every data provider and then combining these results for computing the ten nearest neighbors of the whole real estate data set. The Union operator could be carried out by one of the cycle providers that carry out the low-level *nn_10* operations or by a dedicated cycle provider in order to increase (pipelined) parallelism. Pure, dedicated cycle providers are also necessary in this example if one of the real estate data providers is not capable (e.g., not enough main memory) or not willing (e.g., for security reasons) to serve as a cycle provider.

2.3 Quality of Service (QoS)

As seen in the real estate example query, query execution in ObjectGlobe can involve a large number of different function, cycle and data providers. A traditional optimizer produces a plan that reads all the relevant data (i.e., considers all real-estate data providers). Therefore, the plan produced by a traditional optimizer will consume much more time and cost than an ObjectGlobe user is willing to spend. In such an open query processing system it is essential that a user can specify quality constraints on the execution itself. These constraints can be separated in three different dimensions:

Result: There are several important properties of a query result a user should be able to specify. For example, a user may want to restrict the size of the result set returned by his/her query in the form of a lower or an

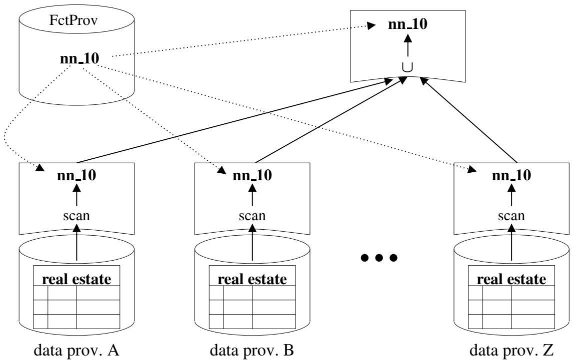


Fig. 4 Parallel Execution in ObjectGlobe

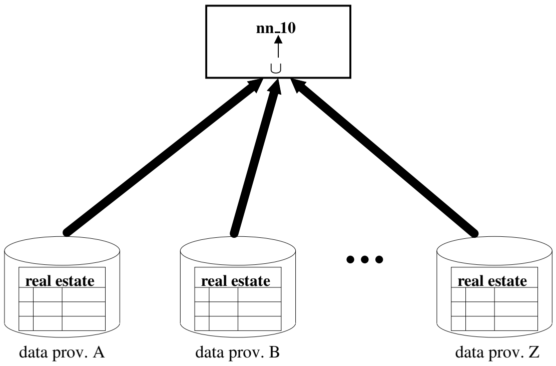


Fig. 5 Execution in a Middleware System

upper bound (an upper bound corresponds to a stop after query [CK98]). Constraints on the amount of data used for answering the query (e.g., at least 50% of the data registered for the theme "real estate" should be used for a specific query) and its freshness (e.g., the last update should have happened within the last day) can be used to get results which are based on a current and sufficiently large subset of the available data.

Cost: Since providers can charge for their services in our scenario, a user should be able to specify an upper bound for the respective consumption by a query.

Time: The response time is another important quality parameter of an interactive query execution. A user can be interested first, in a fast production of the first answer tuples and second, in a fast overall execution of the query. A fast production of the first tuples can be important so that the user can look at these tuples while the remainder is computed in the background.

In many cases not all quality parameters will be interesting. As in real-time systems some constraints could be strict (or hard) and others could be soft and handled in a relaxed way.

An overview of processing a query in the context of our QoS management is depicted in Figure 6. The starting point for query processing in our system is given by the description of the query itself, the QoS constraints for it and statistics about the resources (providers and communication links). As shown in the figure, QoS constraints will be treated during all the phases of query processing:

- The optimizer which generates the query evaluation plan (QEP), looks for data providers which will contribute enough and sufficiently current data so that the constraints regarding completeness, cardinality and freshness are fulfilled.
- During optimization we estimate the quality parameters of all enumerated sub-plans and plans. Only a plan which fulfills all constraints is executed and its plan description will be annotated with the quality estimations and resource requirements for every sub-plan. Additionally, if the optimizer can find equivalent alternatives for resources used in the query evaluation plan, these are also annotated in the plan. It should be noted here that the cardinality is estimated by the normal selectivity estimation mechanisms of the optimizer, for example, by the use of histograms.
- During the plug phase, sub-plans are distributed to cycle providers, functions are loaded from function providers and connections to data providers are established. When a sub-plan of a query uses the services of a specific provider, it is checked, if the resource requirements resulting from the quality constraints for that sub-plan can actually be satisfied by this provider. For instance, the load of a cycle provider is checked before the execution of a sub-plan is started. If the load is too heavy, the sub-plan is demoted to another cycle provider or the query is aborted. Other actions admission control might carry out are to refine the priorities of queries or, in extreme cases, to call the optimizer in order to re-optimize a (sub-) plan [IEE00].
- During query execution, estimation errors by the optimizer and fluctuations regarding resource availability for, e.g., CPU time or network bandwidth jeopardize the constraints on the quality parameters. Therefore, a monitoring component traces the current status of these parameters for every relevant sub-plan of the query. If this component detects a potential violation of the quality constraints for a sub-plan, it first tries to adapt the sub-plan so that it will meet its constraints again, or if this is not possible, it will abort the execution of this sub-plan. The plan adaptations during the instantiation phase can be performed rather easily, because the plan is not instantiated yet. The adaptations for the execution phase have to conserve the work already done by the plan until the adaptation was triggered. Thus, these adaptations are more complex than those for the instantiation phase.

In summary, the optimizer first generates a query evaluation plan whose estimated quality parameters are believed to fulfill the user-specified quality constraints of the query. For every sub-plan the optimizer states the minimum quality constraints it must obey in order to fulfill the overall quality estimations of the chosen plan and the resource requirements which are believed to be necessary to produce these quality constraints. If, during the plug phase, the resource requirements cannot be satisfied with the available resources, the plan is adapted or aborted. The QoS management reacts in the same way, if during query execution the monitoring component forecasts an eventual violation of the QoS constraints.

2.4 Privacy and 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 privacy and 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 external query operators are therefore executed in a protected area, a so-called *sandbox* (Section 4.4).

Privacy and Confidentiality: Data and function code that is processed in the ObjectGlobe system is protected against unauthorized access and manipulation. The communication streams between ObjectGlobe servers are protected using the well-established secure communication standards SSL (Secure Sockets Layer) [FKK96] and/or TLS (Transport Layer Security) [DA99, TLS] for encrypting and authenticating (digitally signing) messages. Both protocols can carry out the authentication of ObjectGlobe communication partners via X.509 certificates [HFPS99, PKI]. Furthermore, confidential information or function code is protected from being transferred to untrusted cycle providers by enforcing an authorization scheme on the flow of data and operator code specified in the site annotations of the query plan.

User Authentication/Anonymity: 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. Cycle providers can also require authenticated external operators to restrict the function 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. As already mentioned, providers can also constrain the information (function code) flow to ensure that only trusted cycle providers are used in the query execution plan. 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, which means, that they must be incorporated in its lookup service.

2.5 Comparison to Other System Architectures

Distributed database systems have been studied since the late seventies in projects like System R*, SDD-1, or Distributed Ingres. A survey of existing distributed query processing techniques studied in these projects is given in [Kos01]. ObjectGlobe shares with all these projects the vision that a distributed 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 architecture of ObjectGlobe is more general than that of a traditional system like System R*. In a traditional system, every site acts as a data and cycle provider which executes built-in query operators; obviously, ObjectGlobe supports such a scenario as well. In addition, ObjectGlobe provides the flexibility to integrate external operators and a large number of non-database (legacy) data sources.

Today, external operators and/or legacy data sources are typically integrated using a middleware architecture; examples are Garlic [C+95] from IBM, Information Manifold [LRO96], TSIMMIS [PGGMU95], DISCO [TRV98] or Tukwila [IFF+99]. Again, ObjectGlobe's architecture is more flexible, resulting in better performance. Let us see how our example query shown in Figure 2 would be processed in a middleware system. As shown in Figure 7, middleware systems can only exploit the (limited) query processing capabilities that are hard-wired into the (legacy) data sources. If new operators are needed, such as *wrap_S* and *thumbnail*, these operators are executed at a central middleware site. This is also true for distributed middleware systems like Amos*II* [JKR99], because the corresponding server processes are restricted to the mediator's capabilities and cannot be extended by dynamically loaded mobile code. This means, that only specific servers, which can be prepared by a user in advance, can execute his/her application specific code. In Figure 4 the ObjectGlobe version of the nearest neighbor example plan is depicted. In contrast to the traditional execution plan of middleware systems as shown in Figure 5 the ObjectGlobe plan, which

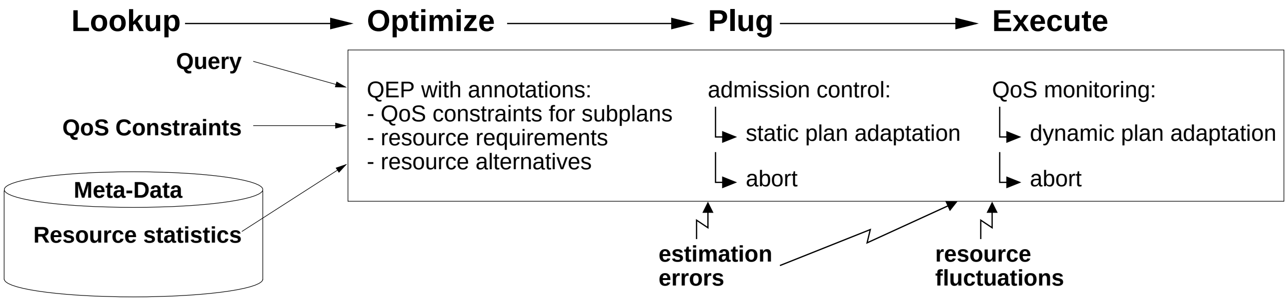


Fig. 6 The Interaction of Query Processing and QoS Management

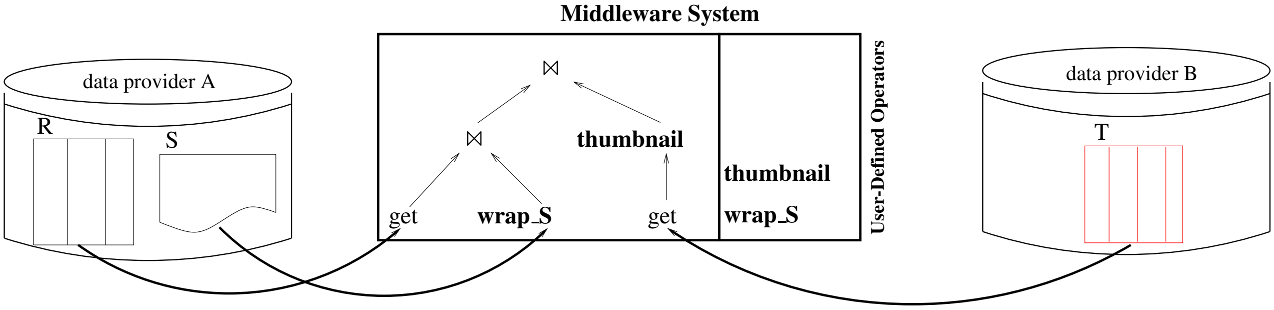


Fig. 7 Query Evaluation in a centralized Middleware System

uses dynamic operator loading, can exploit parallel execution of several nearest neighbor operators and causes much less network traffic. As a result, a middleware system incurs high communication costs for shipping the data to the middleware; i.e., for data shipping [FJK96]. ObjectGlobe helps reduce such communication costs by allowing to execute new and external query operators at or near the data providers.

Various aspects of the ObjectGlobe project have already been studied in other projects. 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. Mariposa also has some notion of QoS, but we consider user-defined quality constraints during all phases of query execution, whereas Mariposa tries to obey these constraints only during its plan fragmentation step, which takes place after optimization. We believe, that this is not sufficient in such an Internet-wide open query processor.

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.

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]. The UDDI standard [UDD00] defines a framework for the management of meta data about electronic commerce Web services. 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]. ObjectGlobe's lookup service also bears some similarity with X.500 [CCI88] and LDAP directory services [WHK97]. 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 this respect, ObjectGlobe's lookup service is similar to [MRT98]'s WebSemantics project which uses Web documents to publish the location of components (wrappers and data sources) and a uniform query language to locate data sources based on this meta-data and to access the sources.

In other lines of work, researchers have tried to "query the Web" using languages like WebSQL [MMM97, KS98]; 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.

3 Generating Query Plans

In this section, we show how ObjectGlobe produces a plan for a query. In particular, we describe the ObjectGlobe *lookup service* that finds relevant resources for a query and the parser and the optimizer that try to find a good plan to execute 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.

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. Providers are registered before they can participate in ObjectGlobe. In this way, the information about available services is incrementally extended as necessary. A similar approach for integrating various business services in B2B e-commerce has been proposed recently in the UDDI standardization effort [UDD00].

We expect the registration of providers' services to become a similar market as the market for the providers themselves. So, someone interested in using a service will register this service; service providers themselves need not necessarily do this on their own. For example, wrapper developers are of course interested in registering data sources for which they have written the corresponding wrappers. Such an incremental schema enhancement by an authorized user is possible in the ObjectGlobe lookup service just as in any other database system. This means, that an ObjectGlobe system is normally not tailored for a specific data integration problem, but can dynamically be extended with new data, cycle, and function providers by augmenting the meta-data of its lookup service.

The ObjectGlobe parser and optimizer consult the lookup service in order to find relevant resources to execute a query and obtain 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.

3.1.1 ObjectGlobe's 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. A collection is either a materialized partition conforming to ObjectGlobe's internal nested relational format or a virtual collection, i.e., an Internet data source transformed into the collection's recorded schema by a wrapper. Collections are associated to a specific *theme*. A theme describes a special concept with a set of terms, called *attributes*. A theme's attributes can be viewed as the union of all attributes meaningful for the theme. Queries are formulated over the themes and their attributes. Integration of a new data source is achieved by registering it as a new collection and associating it to a theme. So collections can be seen as horizontal (possibly overlapping) partitions. The attributes provided by the new collection must be a subset of the attributes defined by the associated theme. Currently ObjectGlobe uses a non-hierarchical set of themes, but more complex ontologies [BCV99] could be added on top of our flat theme structure. As an example, `www.HotelBook.com` and `www.HotelGuide.com` provide different collections which are associated to the theme *hotel*.
Furthermore, the lookup service stores binding patterns of a collection, statistics about a collection like histograms for estimating the selectivity of simple (i.e., non-external) predicates, and information about replicas (i.e., mirrors) of a collection, which could be 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. The load on the cycle provider regarding CPU power and available main memory is stored as a function of time and likewise we store the latency and bandwidth information for the network links between cycle providers.

function provider: The name and signature of each query operator is recorded. Furthermore, formulas to estimate the consumption of CPU cycles, main memory, disk space, and the selectivity for each query operator are kept by the lookup service. These formulas use a set of parameters which describe the characteristics of the executing cycle provider (e.g., the available CPU power/main memory) and the input data for a specific application of this operator.
ObjectGlobe differentiates between *iterators* like join or display and *transformers* such as *thumbnail*. (In addition, ObjectGlobe has also 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.

authorization information: the lookup service maintains authorization information which is obtained from the providers and indicates which data may be processed at which cycle provider and by which query operator. To guarantee privacy and confidentiality, the providers can also restrict the flow of information (and code) in order to prevent data (and functions) from being processed on untrusted cycle providers. Following the ObjectGlobe authorization model, it is possible to specify positive and negative authorizations [RBKW91,BJS99]. Also, it is possible to group collections, functions, and cycle providers into "authorization classes"—using role-based authorization [SCFY96]—in order to reduce the overhead of maintaining and processing this information in the lookup service.

Appendix B shows an example `RDF` document that can be used by a data provider to register a *hotel* collection. The meta-data kept in the lookup service can be outdated or incomplete. It is possible, for instance, that a data provider revokes the privilege of some cycle providers to process its data without notifying the lookup service; as a result, the execution of a query might fail due to an authorization violation which is detected at execution time. ObjectGlobe relies on data, function, and cycle providers to notify 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 are eventually excluded from the ObjectGlobe federation. As an alternative, [CZH+99] proposes to use a *time-to-live* scheme; in that scheme, providers must periodically contact the lookup service if they want to continue to remain in the federation.

3.1.2 Using the ObjectGlobe Lookup Service As mentioned before, data, function, and cycle providers are registered by generating `RDF` documents describing their services. We use `RDF` because it is very flexible and a WWW standard for describing resources [BG99]. Typical collections, 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. An `RDF` document 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 9 shows how to ask the lookup service for all collections that supply data for the *hotel* theme. More specifically, the query of Figure 9 asks for *hotel* collections which have *city*, *address*, and *price* attributes and the query asks for the `uniqueId` of the collection (used to identify replicas) and information about all *attributes*. (The "?" in the query is an *any* operator.) The result of this query is shown in Figure 10; 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 a distributed component of the ObjectGlobe system and it is implemented in a hierarchical architecture (Figure 11). A relational database system serves as basic data storage, mainly for the advantages in robustness, scalability, and query abilities. Meta-data (i.e., `RDF` documents) are mapped to 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. Lookup service clients, 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. Figure 11 shows the following lookup service components:

Providers: Cycle, data, and function providers *register* their services and resources at one of the backbone meta-data providers (MDPs).

Meta-data Providers: The backbone of the MDPs contains global meta-data, usable by everyone on the Internet. The data registered at the backbone meta-data providers is kept consistent, i.e., the meta-data is replicated between the MDPs of the backbone. If some data is updated at one MDP, the update is propagated to the other MDPs of the backbone.

Local meta-data repositories: Basically, a local meta-data repository (LMR) caches meta-data of an MDP. If this data is changed (at an MDP), all caching LMRs are notified. Rules (similar to queries) are used to specify which meta-data should be cached. An LMR *subscribes* to an MDP and registers its subscription rule set. The MDP uses the rule set to determine the local meta-data repositories to which newly registered meta-data must be *published*. It is also used to forward update notifications to LMRs.

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

```
select price, address
from hotel
where city='New York'
```

Fig. 8 SQL Query

```
search Partition d
select d.uniqueId, d.attributes.*
where d.theme.name="hotel"
  and d.attritutes.?.topic="city"
  and d.attritutes.?.topic="address"
  and d.attritutes.?.topic="price"
```

Fig. 9 Example Search Query

```
<collection>
  <uniqueId>4711</uniqueId>
  <attribute topic="city" domain="String"/>
  <attribute topic="price" domain="Integer"/>
  <attribute topic="address" domain="String"/>
</collection>
```

Fig. 10 Example Search Result

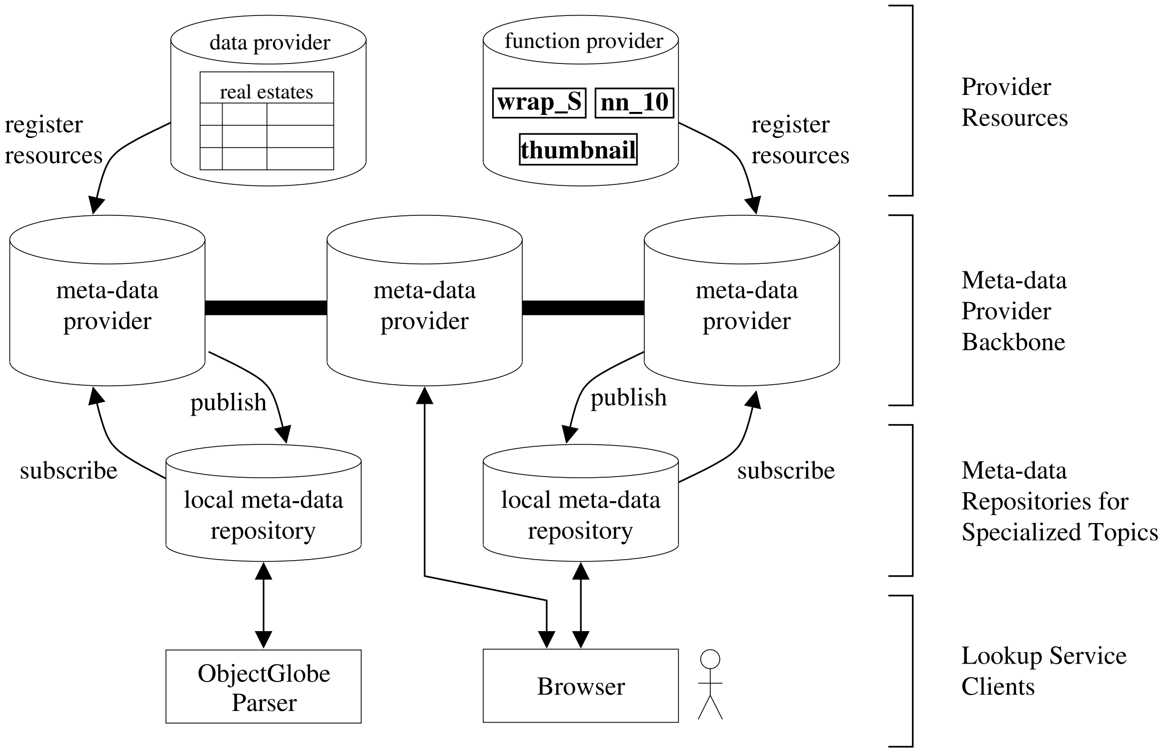


Fig. 11 The Architecture of the Lookup Service

Additionally, an LMR stores local meta-data that should not be accessible to the public. Therefore this meta-data is not forwarded to any MDP.
For efficiency reasons (note that the meta-data lookup is a part of the query optimization) search requests are processed by LMRs only. An LMR uses only local and cached meta-data to evaluate a search request. Only explicit MDP requests are forwarded to an MDP.

Lookup Service Clients: Clients access the lookup service by connecting to an LMR and stating queries using the lookup service's query language. Figure 11 depicts two clients, the ObjectGlobe parser and an end user browsing the meta-data of an MDP and an LMR.

Typical subscribers to an MDP will register hundreds of rules. The set of all registered rules is called the *subscription rule base*. If new meta-data is registered, updated, or deleted at an MDP, all registered rules must be evaluated. To improve performance, an MDP applies a prefilter algorithm that takes the modified meta-data and efficiently determines a superset of the rules that are affected by the modification. In a second step, all rules of this superset are evaluated incrementally using only the modified meta-data. Only some special rules require additional, unmodified meta-data to be included in the evaluation. All basic parts of the prefilter algorithm are mapped to SQL queries executed by the RDBMS used as data storage. The scalability of RDBMSs regarding a great amount of data and multiple queries at a time is used by the lookup service's prefilter algorithm to gain scalability in terms of a large database and a multitude of subscription rules stemming from the many LMRs. Additionally, the lookup service's architecture itself is scalable in terms of the number of users by adding additional LMRs when necessary. A forthcoming paper describes this part of the lookup service in more detail [KKKK01].

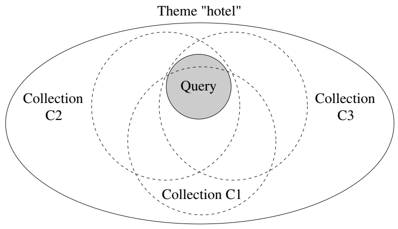


Fig. 12 Relationship of Theme, Collections, and Query Attributes

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., collections, functions, and cycle providers). The parser aborts the processing of a query if for some part of the query, no resources can be found. Relevant collections are found using the *themes* and *attributes* specified in a query. All themes used in the query's `FROM` clause and their corresponding attributes used in the `SELECT` and `WHERE` clauses define the query's schema. The relationship between the attributes used in a query, the attributes recorded for collections in the lookup service, and the attributes of a theme are depicted in Figure 12. For every *theme* referred to in the query, the parser queries all *matching* collections from the lookup service; a collection matches if it is associated to the requested theme and provides a superset of all attributes used in the query. For example, assume the SQL query given in Figure 8. From this query the parser determines a schema consisting of the attributes *hotel.city*, *hotel.address*, *hotel.price*, represented by the gray filled circle in Figure 12. To find all relevant collections the parser queries from the lookup service all collections associated to the *hotel* theme (collections C1, C2, and C3) and providing at least the attributes *city*, *address*, and *price* (only collections C2 and C3). The resulting search request to find relevant collections for the query of Figure 8 is given in Figure 9.

As Figure 12 shows, collections may provide more attributes than are actually used in a query. In the execution phase, the schema of a collection is projected to the schema required by the query execution plan. So, in Figure 12, the operator used to access collection C2 will not return all attributes represented by the dashed circle C2, but only the attributes of the intersection of the sets C2 and Query (the attributes *city*, *address*, and *price*).

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 the 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 13 shows how the authorization and applicability information is represented as a *compatibility matrix* for the collections, functions, and cycle providers of the example of Section 2.2. For each relevant data 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

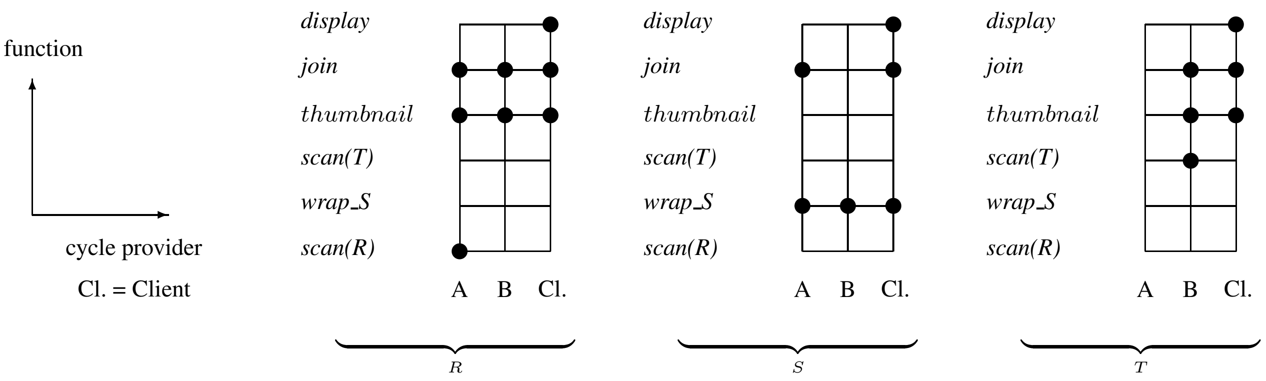


Fig. 13 Compatibility Matrices for the Example of Section 2.2

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, unlike 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 specify quality parameters for the query execution itself as described in Section 2.3. For example, if the user's upper bound for the costs of a query is 10 € and the optimizer does not find a matching plan for this constraint, the user is informed about this fact and no query execution takes place.

The optimizer enumerates alternative query evaluation 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 quality 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:

Quality of Service Model The support for user defined QoS constraints on queries makes it necessary to use a more general measurement model for query plans than the usually implemented cost models. In contrast to the traditional one dimensional cost assessment our *QoS model* uses a separate dimension for every quality parameter, like response time, monetary cost, and result cardinality. All these dimensions span a space, which we call QoS space, and the user-defined constraints determine an area in that space, which we call QoS window. This is shown in Figure 14 for a (simplified) three dimensional QoS space. During optimization every enumerated plan is mapped to a point in that QoS space by estimating the value for every quality parameter, which appears in the quality model. Only plans which lie within the QoS window fulfill the user constraints. As shown in [GHK92], where we borrowed the basic ideas for multi-dimensional optimization, pruning can only work with a partial order in such a setting. This is depicted in Figure 15, where we restricted the QoS space even further to only two dimensions in order to simplify the illustration. The figure shows, for example, that the plan `P1` is superior regarding time and cost consumption to the plans `P4` and `P5`. Although `P1` produces the query result faster, its execution is cheaper than the execution of `P4` and `P5`. `P1`, `P2`, and `P3` are incomparable, but only `P1` and `P2` are candidate plans, because `P3` lies outside the QoS window. The arrows emanating from these incomparable plans mark the area in the QoS space, which is dominated by the respective plan. The plan `P4` lies inside the QoS window (i.e., the plan fulfills the user constraints), but it is no candidate plan, because it is dominated by the plans `P1` and `P2` both of which are superior to `P4` in *all* dimensions of the QoS space. Thus, `P1` and `P2` are the only plans "surviving" the pruning. The decision between `P1` and `P2` is made according to a heuristic, which chooses the plan with the largest, minimum normalized distance to any of the borders of the QoS window.

To estimate the quality parameters of a plan the optimizer relies on the statistics and measurement functions registered in the lookup service. In the absence of such statistics, the ObjectGlobe optimizer will *guess* (i.e., use

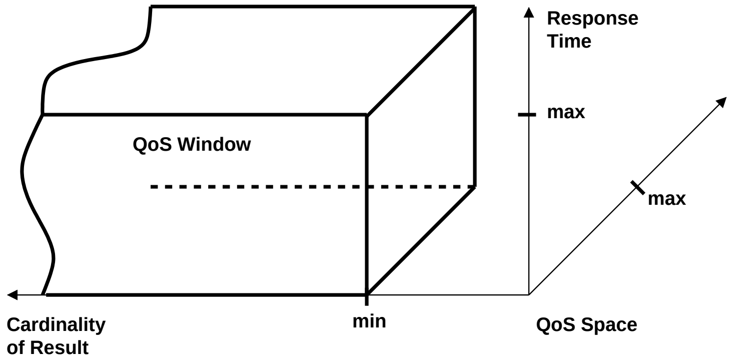


Fig. 14 The QoS Space and the QoS Window.

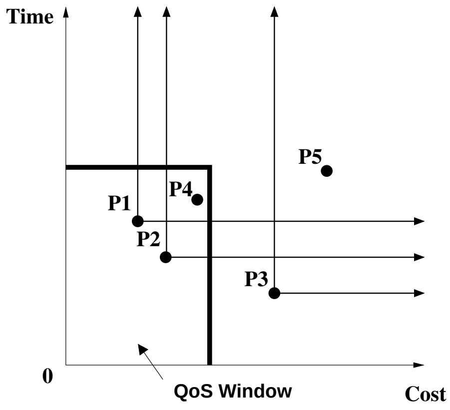


Fig. 15 The Partial Order for Plans.

default values), just as any other optimizer. Work on assessing plans in distributed and heterogeneous query processors without explicit knowledge of the involved data sources has been reported in [ROH99] and we are extending our framework along the lines of this work. For a more detailed description of QoS in ObjectGlobe the interested readers are referred to [BKK01].

Optimization Goal The optimization goal of our optimizer is closely related with the goal of the QoS management component. There are two obvious goals for our QoS management component and the first one concerns query optimization:

- The percentage of successful queries, whose quality constraints could be fulfilled, should be maximized. This percentage is calculated based on the overall number of queries which are issued and not only on the number of those for which a constraint compliant query plan could be determined.
- The execution of queries which cannot fulfill their QoS constraints, should be stopped as early as possible.

A query can only meet its quality constraints, if it gets a sufficiently good service from all involved providers. The difficulty in achieving a high percentage of QoS compliant queries is to find at optimization time a query plan that uses providers which can provide for a sufficiently good service at execution time. The optimizer uses estimates about the providers to construct such a query plan and the question is now, whether these estimates also hold during execution. We cannot always work with resource reservations at optimization time because the administrative overhead and the inherent reduced resource utilization are not acceptable for all providers. In our approach we exploit that different queries often have different demands at specific providers (e.g., batch queries in contrast to interactive queries). For every involved provider we explicitly state these demands in the form of resource requirements and quality constraints in the query plan. During the plug- and execution phases, we can use this information in order to check, if the demands of a query are really affected by other queries. Every query which seems to miss its quality constraints, tries to get a greater share on the resources of the provider at the expense of queries which can work sufficiently with a smaller share. If such an adaptation is not possible and the query will not fulfill its QoS constraints our second goal demands to stop the query in order to save the time and money of the user.

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 (13). The matrix of a join plan which is composed of two sub-plans is generated by ANDing the two compatibility matrices of the two subplans, 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 an estimated value for one of its quality parameters 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.

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 [KS00].

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 *cycle-provider* 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) secure communication paths are established using secure protocols (Section 2.4).

4.2 Authentication and Authorization

If a provider restricts the use of its resources and therefore requires some kind of authentication of users the authentication information will be part of the query plan (again, as part of an annotation). Two possible authentication schemes are supported. (1) A user can provide a password. The password is used to generate a secret key (using the PKCS#5 Password-Based Encryption Standard [RSA99]) which is afterwards used to calculate a MAC (Message Authentication Code) of the query plan and some additional data. (2) The user possesses a valid X.509 certificate [HFPS99, PKI]. The certificate is used to calculate a digital signature of the query plan and some additional data.

One problem remains. What if a data provider does not support one of these schemes, i.e., requires the password in plain text? The password is included (as authentication information) in the query plan. The wrapper accessing the data provider extracts the password and passes it to the data provider. To keep the password secure it is encrypted with the public key of the cycle provider that executes the wrapper. So no other cycle provider is able to access the plain password.

Authorization is carried out by the individual providers when a query is instantiated. Each provider autonomously decides if it allows the local execution of the query plan depending on the local policy. Most providers will delegate this decision to a local security provider which is included in the ObjectGlobe system. Data providers may also have their own security system (as most DBMSs have) that they can use instead of the ObjectGlobe security provider.

The security provider uses a role-based access control (RBAC) model [SCFY96] to specify authorization rules. RBAC distinguishes between users, roles which are assigned to users and permissions which are assigned to roles. ObjectGlobe provides permissions for allowing or denying access to a relation (i.e., executing a wrapper), loading and executing an operator and using a cycle provider (i.e., execute a query plan at the cycle provider).

4.3 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 `re-open` 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 *outer 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.

4.4 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 16. 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., 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 (to be accurate every external operator runs within its own namespace to avoid problems with name clashes and version mismatches). This way it is guaranteed, that they cannot illegitimately exchange information via covert channels ("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. Since an external operator could abuse the connection to a function provider as bidirectional communication channel, all (non built-in) classes required by an external operator must be combined into a JAR[3] file. This archive file is loaded and cached by a class loader and the connection to the function provider is closed. All requests to non built-in classes must point to classes in the cached JAR file otherwise they are rejected as illegal. Schematically, the name space separation and the class loaders are illustrated in Figure 17(a).

Some user-defined query operators may 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

[3] JAR (java archive) is a platform-independent file format that aggregates many files (compressed) into one (like ZIP) and is supported by the Java Runtime Environment.

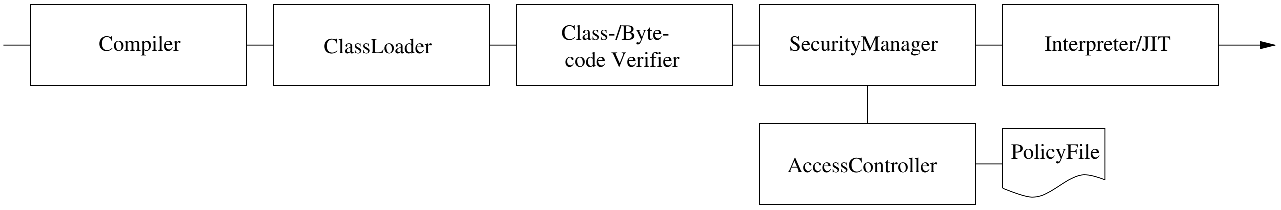


Fig. 16 Java's Five-Layer Security Model

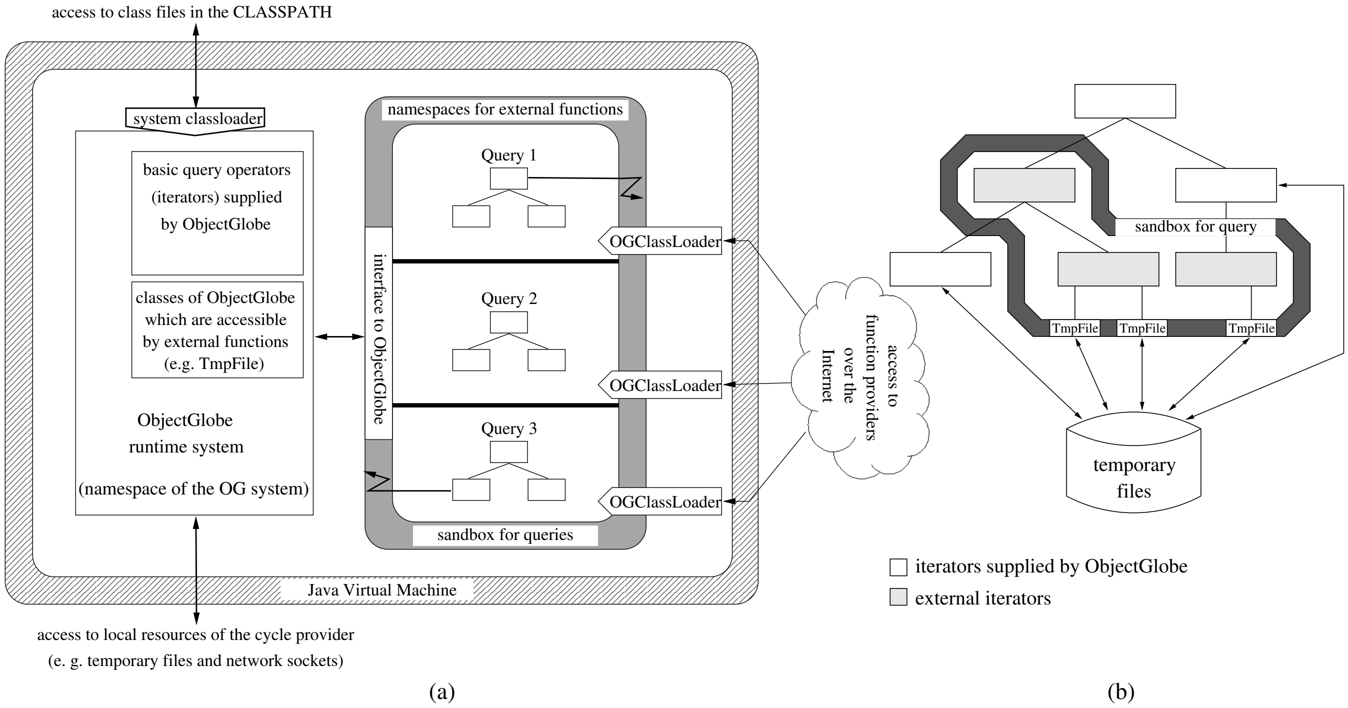


Fig. 17 (a) Security of Dynamically Loaded Code, (b) Extending Privileged Access Rights to User-Defined Operators

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 17(b).

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 can 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 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 intruders. We developed a (system dependent) java library based on the Java Virtual Machine Profiler Interface (JVMPI) [Sun99], an (experimental) interface provided by Java Release 2. This library keeps account of memory and CPU usage of external operators, other resources like the number of bytes written to secondary memory can be determined using pure Java.

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.

4.5 Monitoring the Progress of Query Execution

As we have mentioned in Section 2.3, the optimizer determines for each sub-plan of a query threshold values for the time and cost consumption and the cardinality of intermediate results which need to be met by the query execution in order to fulfill the user-defined quality constraints. Wrong estimates or resource fluctuations can cause overdrawn thresholds, which probably result in a violation of the quality constraints. Therefore, our QoS management component monitors the query execution in order to detect and to react on potential quality violations.

As an important sub-task of this monitoring, we have to check, if a query still makes any progress at all. The execution of a distributed 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.

4.5.1 Monitoring the Liveliness of Query Execution Each ObjectGlobe server uses a dedicated thread (we call it the *monitor thread*) for detecting timed-out 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 expired, the monitor thread removes the timeout object from 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 below the blocked node will normally fail. The operators above it 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.

So far we have not mentioned where the timeout objects come from. These objects are created by a special type of operator, the *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 task is to observe the progress of the actions performed by the sub-plan beneath. 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 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 18.

4.5.2 Monitoring the QoS of Query Execution Monitor operators are not only used for observing the liveliness of a query execution, but also measure the current status of the quality parameters of the execution. As we have mentioned in Section 3.2 the quality model, which is used for assessing query execution plans during optimization, produces for every sub-plan an estimated value for each quality parameter, like response time, cost and cardinality. These estimates can be seen as balances of accounts, which can be positive or negative. A balance of 500 for the response time account and 100 for the cost account, for example, tell the monitor operator that the execution of the sub-plan beneath it is allowed to last 500 time units and may cost at most 100 monetary units. For a cardinality account with a balance of -700 we can infer, that the sub-plan should at least produce 700 result tuples. An example account configuration is shown in Figure 19 for a query searching for real estate, which are close to a bigger city (predicates are not shown in the figure).

During query execution monitor operators keep track of the number of tuples produced, the time and cost consumption of the execution and some rates like cost or time consumption per produced tuple. These rates are used for

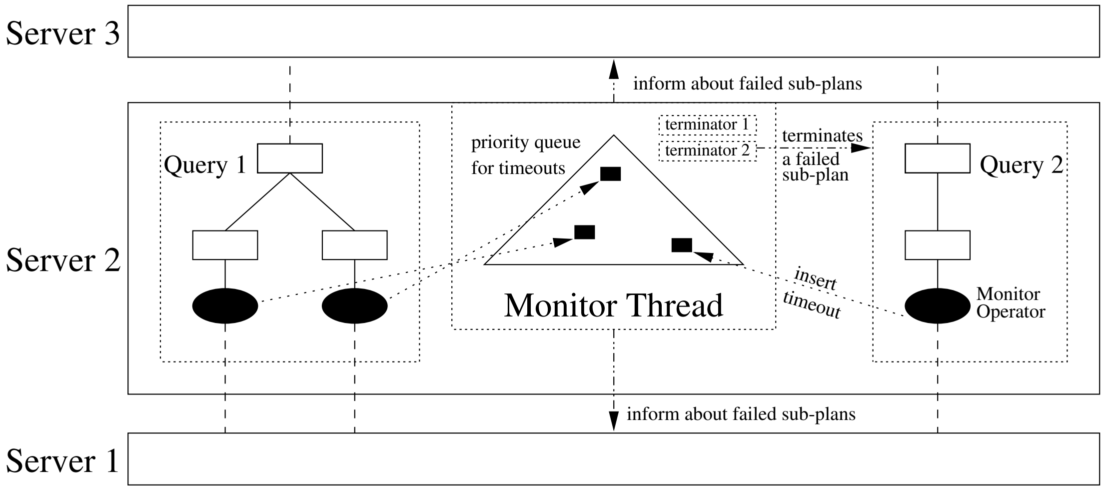


Fig. 18 The Architecture of the Monitor Component.

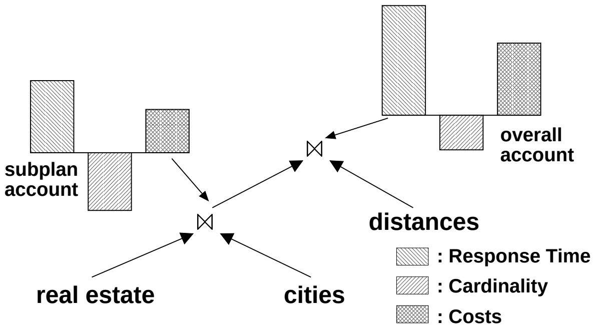


Fig. 19 QoS Accounts of a Query Plan.

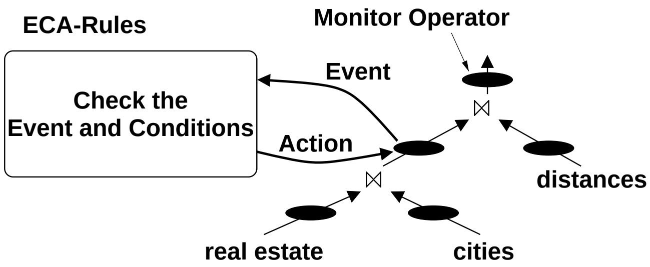


Fig. 20 Feedback Loop for QoS Adaptations.

projecting the past development of the quality parameters into the future. For example, the formula

$$T_N + R_{TN}(C_E - C_N)$$

uses the time consumption for the production of all the tuples so far (T_N), the time consumption per produced tuple (R_{TN}) and the estimated and current result cardinality, C_E and C_N, to compute an estimate for the overall time consumption of the sub-plan. If this estimate exceeds the balance of the corresponding account, we expect that the overall execution of the sub-plan will eventually violate the constraints for that quality parameter.

The quality constraints of a query execution are mainly jeopardized by inaccurate estimates during the optimization phase and resource fluctuations in the distributed environment. To ensure a prompt reaction of our forecasting mechanisms in a changing environment we do not base the computation of the consumption rates on the whole history of the query execution, but on a smaller, current window of it, i.e., we use a moving average computation. A prompt signaling of a quality loss event helps our QoS management component in applying a corrective adaptation of the execution early, so the adaptation has more time to show an effect. If no adaptation seems appropriate for a quality loss event, we also profit from an early signaling, because we can stop the execution at an early stage, when the consumption of time and cost is still low.

In the following we give an overview of the adaptations which we apply on query execution plans. Naturally, the kind of jeopardized quality parameter determines the set of useful adaptations.

Prevention of Response Time Violation If a sub-plan seems to miss its constraints on the response time, we can use adaptations on the resource or the application level. On the resource level, for example, we can change the priority of the respective thread, the main memory allotment for the respective operators or we can renegotiate the network service quality, if the used network supports itself QoS handling like an ATM network. However, more promising are adaptations on the application level like the activation of compression at runtime for the data sent through a network link or the movement of complete sub-plans together with their state from one cycle provider to another—again, during the runtime of the query. For example, if a monitor operator detects that a sub-plan suffers from too small a percentage of CPU time which is available for it on a cycle provider, the QoS management component can decide to move this sub-plan with all its state information to another, better suited cycle provider. The remainder of the sub-plan's work is then performed on the new cycle provider. All the other sub-plans of the same query above and beneath that sub-plan are not affected by this move operation, because the relevant communication links between the sub-plans are disconnected and reestablished automatically by the runtime system of our query processor.

Prevention of Cardinality or Completeness Violation If the cardinality constraints or the completeness constraints are in danger, we can use an adaptation accomplished by our union operator, which establishes a new branch at runtime. This means, that we integrate additional data sources in the query execution, which were not involved in the original query execution plan. Of course the information about these additional data sources was appended to the plan during the optimization phase.

Prevention of Cost Violation If the costs of a sub-plan seem to exceed the corresponding limit, we can try to reduce the amount of processed data, by stopping input plans before they are finished. Other ways for reducing cost consumption are the movement of a sub-plan to a cycle provider, which charges less money for the execution, or the exchange of externally loaded functions, like thumbnail encoders, with versions, that produce, for example, a result with a reduced quality, but with less effort.

The application of an adaptation depends on a number of conditions, as described above, and one must also see, that an adaptation, which tries to remedy a pending quality loss in one parameter, could make the situation worse for another parameter. Therefore, the application of adaptations is controlled by an event-condition-action rule set, which is part of the feedback loop of the runtime QoS management. This situation is sketched in Figure 20.

5 Usage of an ObjectGlobe Federation

In the former sections we described the techniques used in the ObjectGlobe system; now we deal with more global aspects regarding application scenarios. First we sketch some common network constellations of distributed information systems and the security requirements they impose. After that we give some ideas on how a highly dynamic e-commerce marketplace can be implemented based on the ObjectGlobe system.

5.1 Usage Scenarios and their Security Implications

The applications of an ObjectGlobe system can be distinguished according to the openness of the underlying network. In the following paragraphs we describe three different scenarios with varying levels of openness and the resulting security requirements.

Intranet An Intranet is a controlled network within an organization and therefore access is restricted to a limited group of authorized users, the employees of the company. ObjectGlobe's cycle-, data-, and function providers are located within the Intranet and all query operators are written by employees of the company or bought from trustworthy third party suppliers. Therefore, these operators can be executed in privileged mode, e.g., these operators are granted privileges to access the disk or establish network connections. To avoid that operators are manipulated, they should be signed (authenticated) by a responsible security administrator of the ObjectGlobe system. Extended privileges can then be restricted to these authenticated operators. If there is a need for secure communication (e.g., if there are outposts), ObjectGlobe can establish secure communication channels itself or it can rely on underlying network layers (e.g., hard- or software enabling a virtual private network).

Extranet An Extranet is a network that is used by different companies, e.g., by a company and its suppliers, forming a virtual enterprise. An important example for an extranet is an electronic marketplace. There are many different scenarios how virtual marketplaces can be run, but we assume in this example that the core cycle- and function providers of the marketplace are operated by an independent organization, which is also responsible for authenticating (signing) external operators. Within the Extranet these authenticated operators can be executed with additional privileges. Every participant of the marketplace at least operates a data provider to supply its product catalog and operators to access it, but it can operate additional cycle providers, too. The task of such cycle providers could be to execute external operators developed by the participants themselves, either because the marketplace does not trust the operators or because the participants do not want others to execute their operators to prevent, e.g., decompilation of the operators. As in the Intranet scenario there are several built-in possibilities to achieve secure communication.

Internet The (global) Internet is the most challenging environment. As mentioned in Section 4.4, protecting the sensitive resources of cycle providers is necessary because external operators could contain hostile code. There is a great deal of external operators which are not signed or signed by unknown function providers and, thus, cannot be trusted. With its effective security component ObjectGlobe is able to execute such operators in a protected sandbox, thereby guaranteeing security and stability of the system. Furthermore cycle providers must be protected against denial of service attacks. This is done by monitoring resource consumption of external operators. However, the existing monitoring component can only detect simple overconsumption problems.

5.2 Example Application Scenario: Dynamic Electronic Markets

The ObjectGlobe architecture supports e-commerce in two directions. On the one hand it enables the implementation of current application scenarios on top of heterogeneous data sources. That is, complete integration solutions of heterogeneous DBMSs based on ObjectGlobe's query processing capabilities and wrapper technology can be developed. Even a global information-sharing system can be architected as a dynamic ObjectGlobe federation due to its openness, scalability, and decentralization. As another example, current electronic marketplaces with their demand for integrated product catalogs, access to back-end data sources, and applications can also be implemented with ObjectGlobe.

ObjectGlobe provides for more flexibility: It helps to develop new business models and application scenarios. Fine grained application service providing of application logic in the form of user-defined operators is achieved when providers can charge for these services. Providers can confederate to theme communities in order to offer complete service packages, e.g., relocation, travel, raw material, etc. Also finding the right Internet data source, as one of the main problems in the Internet, could be alleviated by specialized providers offering meta-data or resource descriptions.

Since current e-commerce solutions for electronic marketplaces are mostly designed for the needs of large enterprises, which use complex enterprise resource planning systems (ERP-Systems), the majority of smaller enterprises remain excluded. While in existing solutions mostly all available information is centralized (e.g., product and price information, security information, special buyer-seller arrangements), ObjectGlobe enables to architect open and easily accessible marketplaces.

Enterprises are able to participate in the marketplace, either using complex ERP-wrappers or even simple text input forms, appropriate to their back-office solution. Furthermore, marketplace participants can define their own privacy policy (within certain limits). Private information like prices, availability, conditions, or arrangements need not be stored centralized at the marketplace.

Figure 21 sketches an example marketplace application where a complete service package can be requested. A private individual searches for a real estate and relocation support. In contrast to existing marketplace solutions users need not search for each service separately and join the services themselves to find the overall best/cheapest combination but the ObjectGlobe marketplace provides this using a new query operator. Private information like pricing conditions and availability are not stored centrally at the marketplace but remain at the participants' sites, i.e., under their direct control. For that reason distributed sub-queries provide the requested data.

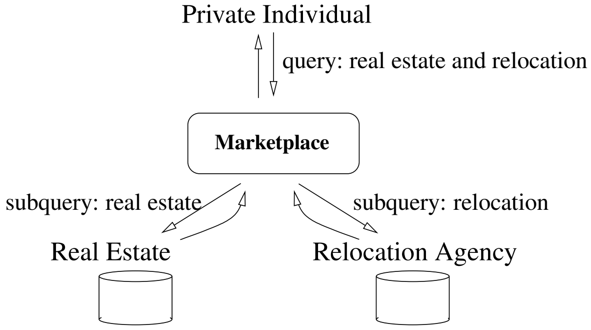


Fig. 21 Relocation Marketplace

Bidders who want to participate only have to register their service, i.e., data description, wrapper, or special query operators, at the marketplace and extend the offered services. For example, a painter can join the relocation marketplace above. Using this approach, new service communities or service portals can be established in the same easy and incremental way. This scenario shows that ObjectGlobe helps in developing a dynamically extensible electronic marketplace, in which new data and service resources can easily be integrated.

6 Performance Experiments

6.1 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. 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.

	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

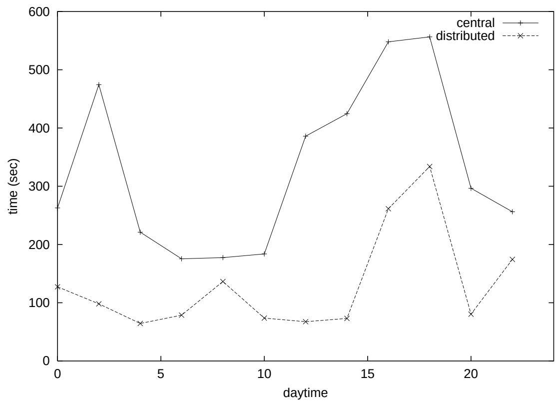


Fig. 22 Centralized versus Distributed Execution of Plans

6.2 Query Execution Times

6.2.1 Benefits of Operator Mobility The following experiment shows the benefits of ObjectGlobe's ability to execute query operators near data sources. We measured the execution time of a query which determines the hotel in Berlin with the greatest number of hotel rooms. The information about hotels is gathered from two Internet sites namely HotelBook (`www.hotelbook.com`) and HotelGuide (`www.hotelguide.com`). To perform this task wrappers were used which first query a list of all hotels in a given city and afterwards query detailed information for every single hotel in this list; according to the query capabilities of the data sources. We measured two different plans for this query, which structurally correspond to the plans shown in Figure 4 and Figure 5, except that we use a group operator instead of a nearest neighbor operator. The traditional one is to execute the wrappers at the client in Passau, the other one which is made available by ObjectGlobe is to execute the wrappers and intermediate group operators at a cycle provider near the data sources. Because it is impossible to execute the wrapper at the hosts serving HotelBook or HotelGuide, we used a host in Maryland for this experiment.

We executed these two plans every two hours in a 24 hour range and as the results in Figure 22 show that there is a clear benefit if the wrappers are executed near the data sources, i.e., at a cycle provider with a good network connection to the data sources. Therefore the latency time is reduced when the wrapper iteratively accesses the HotelBook or the HotelGuide database. This experiment does not demonstrate how parallelism can be used to speed up query execution, because the network costs dominated the CPU costs by far, but performance gains from parallelism can also be achieved with ObjectGlobe.

6.2.2 Costs of Secure Communication The use of SSL sockets [FKK96] and therewith encryption and Message Authentication Codes (MACs) is 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 2 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[4] and across what kind of network the data was sent. The remaining three columns list the times of query 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). 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 much faster than query executions in a WAN environment with unsecure communication). 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.

6.2.3 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. Instead, only the bytecode (rather than the instantiated class object) of an external operator can be cached and this bytecode is cached in a separate ClassFileCache. To measure the overheads of loading an operator from a remote site and from the ClassFile-

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

	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 2 Costs of Secure Communication in Different Network Environments

Cache, we loaded built-in and external operators of different size stored at different locations using our OGClassLoader: built-in operators from disk and external operators from a local function provider in Passau and a remote function provider in Maryland. For external operators, we measure three scenarios: (a) the bytecode is not cached at all; (b) the bytecode is cached in the ClassFileCache; (c) the operator is cached as a class object internally in the OGClassLoader. Scenario (c) is used as a baseline and simulates the behavior of a system without security measures. Figure 23 shows the following effects:

- The costs for the initial loading of a class from disk or network are very high (the +-lines in Figure 23) but can be heavily reduced by caching the class object of built-in operators or caching the bytecode of external operators (the triangle lines).
- Comparing the $\times$-lines (Scenario (c)) and triangle lines (Scenario (b)), we see that the overheads to ensure security are relatively high; compared to the overall costs of query processing on the Internet, however, the overheads for security can usually be neglected (less than a second in all cases).

7 Conclusion

We presented the design of ObjectGlobe, an open, distributed and secure query processing system. The goal of ObjectGlobe is to establish an open marketplace in which data, function, and cycle providers can *offer/sell* their services, following some business model which can be implemented on top of ObjectGlobe. End users and applications can use these services with only little overhead. 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 and QoS at execution time.

The project started about three years ago. An earlier version of this paper was presented on the workshop for e-services [BKK+00] and a first demo was given at SIGMOD 99 [BKK99]. This demo involves two *hotel* servers (i.e., HotelGuide, located in Switzerland, 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 (available on our Web site) can be seen as a simplified e-commerce platform for travel agencies.

Our current implementation is able to run the complete parse – optimize – plug – execute process automatically given a declarative query. While some of the ObjectGlobe components are already quite sophisticated and highly tuned, work on other components, for example, the QoS component, is still in progress and we also need to do some fine tuning regarding the interaction of different components. 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. Besides the work on meta-data management, optimization and QoS, we recently started another project on security and dependability in the ObjectGlobe context. The focus is on (1) runtime resource controlling in order to detect and combat denial of service attacks—and (2) semi-automatic quality assessment of external query operators—e.g., by data flow analysis, automatic stress testing, etc. In an upcoming cooperation project we are building a more generic e-commerce application framework that uses ObjectGlobe as the enabling technology to construct scalable and open virtual marketplaces.

References

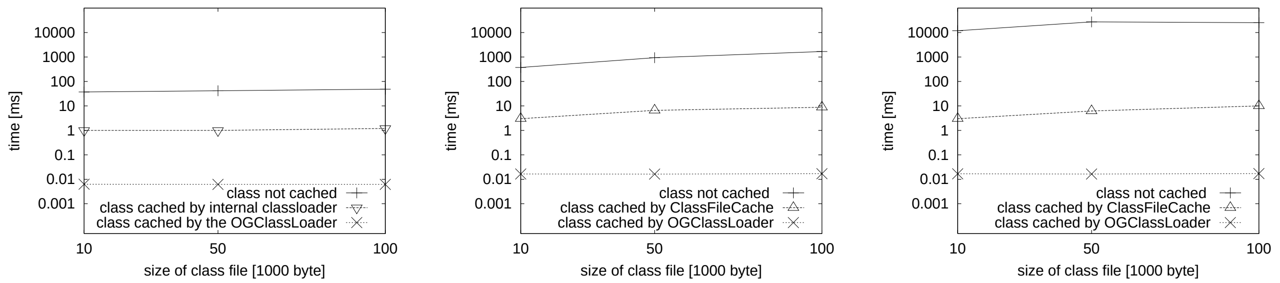
[BCV99] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *ACM SIGMOD Record*, 28(1):54–59, March 1999.

[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.

[BJS99] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140, 1999.

[BKK99] R. Braumandl, A. Kemper, and D. Kossmann. Database patchwork on the Internet (project demo description). In SIGMOD [SIG99], pages 550–552.

[BKK+00] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. In *Workshop on Technologies for E-Services*, Cairo, Egypt, September 2000.



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

Fig. 23 Costs of Loading an Operator by the ObjectGlobe Class Loader

[BKK01] R. Braumandl, A. Kemper, and D. Kossmann. Quality of service in an information economy. 2001. 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.

[CCI88] CCITT International Telegraph and Telephone Consultative Committee. The Directory. Technical Report Recommendations X.500, X.501, X.509, X.511, X.518-X.521, CCITT, 1988.

[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.

[CK98] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 158–169, New York, USA, August 1998.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In VLDB [VLD96], pages 87–98.

[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*, pages 24–35, Seattle, WA, August 1999.

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

[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 SIGMOD [SIG98], pages 414–425.

[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`, November 1996.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–18, San Diego, CA, USA, June 1992.

[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 SIGMOD [SIG98], pages 390–401.

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

[Gur00] G. Gurden. Financial Products Markup Language (FpML). `http://www.fpml.org/spec/fpml-1-0`, September 2000.

[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.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. `http://www.rfc-editor.org/rfc/rfc2459.txt`, January 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In VLDB [VLD97], pages 276–285.

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

[IEE00] Special issue on adaptive query processing. IEEE Data Engineering Bulletin, Vol 23, No 2, June 2000.

[IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution engine for data integration. In SIGMOD [SIG99], pages 299–310.

[JKR99] V. Josifovski, T. Katchaounov, and T. Risch. Optimizing queries in distributed and composable mediators. In *Proc. of the IECIS International Conference on Cooperative Information Systems*, pages 291 – 302, Edinburgh, Scotland, 1999.

[JR99] Vanja Josifovski and Tore Risch. Integrating heterogenous overlapping databases through object-oriented transformations. In VLDB [VLD99], pages 435–446.

[KKKK01] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. Distributed Metadata Management on the Internet. 2001. In preparation.

[Kos01] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 2001. Accepted for publication. To appear.

[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.

[KS98] 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, December 1998.

[KS00] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1):43–82, March 2000.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In VLDB [VLD96], pages 251–262.

[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.

[MRT98] G. A. Mihaila, L. Raschid, and A. Tomasic. Equal time for data on the Internet with WebSemantics. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science (LNCS)*, pages 87–101, Valencia, Spain, March 1998. Springer-Verlag.

[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.

[Pet99] J. Petit. Real Estate DTD. `http://www.4thworldtele.com`, May 1999.

[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.

[PKI] Public-Key Infrastructure (X.509) (PKIX). `http://www.ietf.org/html.charters/pkix-charter.html`.

[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 VLDB [VLD99], pages 599–610.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In VLDB [VLD97], pages 266–275.

[RSA99] RSA Laboratories. PKCS #5 v2.0: Password-Based Cryptography Standard. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf`, March 1999.

[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.

[SAP99] SAP. Business networking in the Internet age. Technical report, SAP White Paper, September 1999. `http://www.sap-ag.de/germany/products/my-sap/pdf/bus_networking.pdf`.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SIG98] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, WA, USA, June 1998.

[SIG99] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Philadelphia, PA, USA, June 1999.

[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 VLDB [VLD97], pages 66–75.

[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.

[Sun99] Sun Microsystems, `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/index.html`. *Java Virtual Machine Profiler Interface (JVMPI)*, 1999.

[TLS] Transport Layer Security (TLS). `http://www.ietf.org/html.charters/tls-charter.html`.

[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.

[UDD00] Universal Description, Discovery and Integration (UDDI) technical white paper. White Paper, Ariba, Inc., IBM Corp., and Microsoft Corp., September 2000. `http://www.uddi.org/`.

[VLD96] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, September 1996.

[VLD97] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[VLD99] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, GB, September 1999.

[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'?>

<plan>
  <iterator id="display" code="iterators.display" cycle-provider="client">
    <iterator id="join1" code="iterators.NestedLoops"
              cycle-provider="client">
      <predicate>Sb = Tb</predicate>
      <iterator id="join2" code="iterators.NestedLoops"
                cycle-provider="alpha">
        <predicate>Ra = Sa</predicate>
        <iterator id="tbscanR" code="iterators.TbScan" cycle-provider="alpha">
          <partition>R</partition>
        </iterator>
        <iterator id="wrapperS" code="wrapper.wrap_S"
                  codebase="functionProvider" cycle-provider="alpha">
        </iterator>
      </iterator>
      <iterator id="thumb1" code="thumbnail" codebase="functionProvider"
                cycle-provider="beta">
        <toThumbNail>picture</toThumbNail>
        <iterator id="tbscanT" code="iterators.TbScan" cycle-provider="beta">
          <partition>T</partition>
        </iterator>
      </iterator>
    </iterator>
  </iterator>

  <provider-information>
    <og-provider id="client">
     <dn-name>C=DE, O=University of Passau, OU=Department
              for Mathematics and Computer Science,
              CN=Mets.fmi.uni-passau.de
     </dn-name>
     <host-dns>Mets.fmi.uni-passau.de</host-dns>
    </og-provider>
    <og-provider id="alpha">
     <dn-name>C=COM, O=A Incorporated, OU=Computing Center,
              CN=alpha.A.com
     </dn-name>
     <host-dns>alpha.A.com</host-dns>
    </og-provider>
    <og-provider id="beta">
     <dn-name>C=COM, O=B Incorporated, OU=Computing Center,
              CN=beta.B.com
     </dn-name>
     <host-dns>beta.B.com</host-dns>
    </og-provider>
    <og-provider id="functionProvider">
     <dn-name>C=COM, O=FctProv Incorporated, OU=Software Development,
              CN=FctProv.com
     </dn-name>
     <code-location>http://www.FctProv.com/forGlobalUse</code-location>
    </og-provider>
  </provider-information>
</plan>
```

B The RDF Registration Code for a Collection

In the sample RDF-description shown below, the relevant information about a data provider can be found enclosed in the `DataProvider` element. It contains information about the name of the provider and a URL with which the data provider can be contacted. The `Partition` element contains information about a collection that the data provider makes available.
At the beginning of the collection description we can find the data provider of the collection, a plain-text description of the content of the collection, the theme (i.e., HotelTheme) this collection is associated with, etc. The element `wrapper` specifies a reference for the wrapper which performs the necessary transformations to integrate the collection into an ObjectGlobe system. More interesting is the content of the `attributes` element. It contains the description of the type of the tuples, given by the collection. In our case the type contains three attributes and for each attribute the name and the type of the attribute are specified. It is possible to insert additional information about attributes which is omitted for brevity.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns="http://www.db.fmi.uni-passau.de/~objglobe/ObjectGlobe-Metaschema.rdf#">

<DataProvider rdf:ID="HotelBook">
  <dataProviderName>HotelBook</dataProviderName>
  <dataProviderUrl>http://www.hotelbook.com</dataProviderUrl>
</DataProvider>

<Partition rdf:ID="HotelBookPartition">
  <dataProvider rdf:resource="#HotelBook"/>
  <partitionDescription>Description of hotels worldwide</partitionDescription>
  <theme rdf:resource="file:/home/objglobe/Themes.rdf#HotelTheme"/>
  <localName>hotelBookPartition</localName>
  <wrapper rdf:resource="file:/home/objglobe/Operators.rdf#HotelBookWrapper"/>
  <uniqueID>4711</uniqueID>
  <cardinality>30000</cardinality>

  <attributes>
    <rdf:Bag>
         <rdf:li><Attribute>
           <topic rdf:resource="file:/home/objglobe/Themes.rdf#cityTopic" />
           <domain rdf:resource="file:/home/objglobe/Themes.rdf#StringDomain" />
         </Attribute></rdf:li>
         <rdf:li><Attribute>
           <topic rdf:resource="file:/home/objglobe/Themes.rdf#addressTopic" />
           <domain rdf:resource="file:/home/objglobe/Themes.rdf#StringDomain" />
         </Attribute></rdf:li>
         <rdf:li><Attribute>
           <topic rdf:resource="file:/home/objglobe/Themes.rdf#priceTopic" />
           <domain rdf:resource="file:/home/objglobe/Themes.rdf#IntegerDomain" />
         </Attribute></rdf:li>
     </rdf:Bag>
  </attributes>
</Partition>
</rdf:RDF>
```