Database Patchwork on the Internet [*]

Reinhard Braumandl Alfons Kemper Donald Kossmann

Universität Passau
94030 Passau, Germany
⟨*lastname*⟩@db.fmi.uni-passau.de

1 Overview

Naturally, data processing requires three kinds of resources:

- the data itself,
- the functionality (i.e. database operations) and
- the machines on which to run the operations.

Because of the Internet we believe that in the long run there will be alternative providers for all of these three resources for any given application. *Data providers* will bring more and more data and more and more different kinds of data to the net. Likewise, *function providers* will develop new methods to process and work with the data; e.g., function providers might develop new algorithms to compress data or to produce thumbnails out of large images and try to *sell* these on the Internet. It is also conceivable, that some people allow other people to use spare cycles of their idle machines in the Internet (as in the Condor system of the University of Wisconsin) or that some companies (*cycle providers*) even specialize on selling computing time to businesses that occasionally need to carry out very complex operations for which regular hardware is not sufficient.

Unfortunately, we are still a far cry from such an open data processing marketplace. What we can already see today is a growing number of data (content) providers, but we can only see a few function providers and virtually no cycle providers. The reason is that there is an asymmetry in the Internet: due to protocols like http or OLE or CORBA, it is possible to move data around in the Internet, but it is not yet as easily possible to move functionality around in the Internet. The only two ways to provide new functionality is (1) to develop Java applets or browser plug-ins, which can be executed at client machines, or (2) to develop middleware systems—in both ways, the data must be shipped to the functionality, which can be prohibitively expensive, if large amounts of data must be processed, and it is not possible to ship the functionality to the data or to ship both the functionality and the data to machines, which are located near the data and particularly suited to carry out the operations.

At the University of Passau, we are currently developing a distributed database system to be used in the Internet. The goal is to ultimately have a system which is able to run on any machine, manage any kind of data, import any kind of data from other systems and import any kind of database operations. The system is entirely written in Java. One of the most important features of the system is that it is capable of dynamically loading (external) query operators, written in Java and supplied by any function provider, and executing these query operators in concert with pre-defined and other external operators in order to evaluate a query. Compared to object-relational database systems, which allow to integrate external data and functionality by the means of extensions (datablades, extenders or cartridges) or heterogeneous database systems such as Garlic [MS97] or Tsimmis [GMPQ$^+$97], our approach makes it possible to place external query operators anywhere in a query evaluation plan as opposed to restricting the placement of external operations to the "access level" of plans. It would, for example, be possible to make our system execute a completely new relational join method, if somebody finds a new join method which is worth-while implementing. Because our system is written in Java, it is highly portable and could be used by data, function and cycle providers with almost no effort. Furthermore, our query engine is, of course, completely distributed providing all the required infrastructure for server-server communication, name services, etc.

2 Example Applications

In the following we describe some applications, which could benefit from the usage of our query engine or just become possible by the use of a system like ours.

Travel Agency: There are a lot of web pages available today, which help to plan an individual journey by

[*]This work is part of the ObjectGlobe project, which is supported by the German National Research Foundation DFG under contract Ke 401/7-1

finding hotel rooms, flights and special events, taking place at the chosen destination. But these web sites normally work by materializing the whole information in a local database. Therefore the given information may be outdated or inaccurate. A further weak point is the lacking possibility to perform joins on several interesting entities. For example, if you want to book a flight to Sydney and a hotel room there, you normally have to consult two different forms and do the join on date on your own. It is also hard work to integrate a new data source for example a car rental agency or a foreign railway company.

Such an application developed within our system would integrate remote data sources for hotels, flights, car rentals and so on by the means of wrappers. Therefore the data would be as up-to-date and as consistent as the original data source is providing them. A classic middleware system could also solve the join problem, but it does not scale well in the Internet. Suppose we want to compute a join between hotels and car rentals in the surroundings of New York, Tokyo and Rome. The middleware system has to fetch all the data from the different sources located near the mentioned cities and performs the join at a second tier server. In our system we could perform a local join at servers located near the mentioned cities and unite the results at an intermediate server or the client. New data sources can be embedded in our system at run time by dynamically loaded wrappers.

Research in Distributed Database Systems: A great handicap in research efforts for distributed database systems is the lack of real world testing possibilities. No research group has the chance to test new plans or operators in an environment with several hundred participating sites distributed over the whole Internet.

Since our query processor is very portable, it would be no problem to install a version of it at computers belonging to interested research groups. All these installations could form one big distributed database accessible by every participating group. Due to the dynamic extensibility, tests and benchmarks with new operators and wrappers would not need any administrative effort by any other group except the one doing the test. But this group will only have to do some configuration on their own installation. The other servers stay untouched by them.

Others: The extensibility of our system at run time can be exploited in quite a large number of applications. For space restrictions we only give some short descriptions of a few of them.

- Dynamically loaded complex predicates could be used for selections on image contents during an access of a remote and large image collection.
- Dynamically loaded wrappers could be used for generating index entries of web pages in Australia for a search engine located in Germany.
- Dynamically loaded operators could be used for integrating new operators in query evaluation plans, for example operators for specialized compression techniques, operators for the efficient processing of "top N" queries, operators for ranking intermediate result objects or new index based join operators.

3 The Implementation of our Query Processor

We used Java as the implementation language for our query processor. Since this language was especially designed for distributed, dynamically extensible applications, it seemed to be the right choice. The key objectives of our query processor, namely portability, security (discussed in Section 3.4) and extensibility, could not have been realized in this extent without Java. This programming language has spread at an enormous speed the last few years. A Java VM is available for nearly every computer platform and with it our query processor can run on these platforms. The extensibility is achieved by the dynamic class loading mechanism of Java and the use of special (Java-) interfaces, which are provided by us. These would be the interfaces, through which dynamically loaded code could interact with the query processor. At the moment we support two interfaces:

- In our query engine there is an interface for objects, which are responsible for performing predicate evaluation on data elements. By the use of this interface users can integrate their own specialized predicates.
- Implementing our interface for iterators in a self-written class is another possibility to be able to incorporate objects into a running server process. In this way new wrappers, user definied functions or iterators can be added to the query engine at run time. All of the example applications mentioned above are using this feature.

In summary our server program (which contains the query processor) is able to provide the following services:

- Exporting data, which is locally managed by the server process itself.
- Providing some core functionality, which can be used by queries or sub-queries running on that server.
- Accepting (parts of) query evaluation plans for execution, in order to exploit idle machine resources.

In the following we give an overview of our implementation.

3.1 Execution Engine Basics

The overall architecture of our query processor is based on the iterator model. We also adopted proposed extensions to this model to support distributed and parallel execution (Send- and Receive-iterators) of query evaluation plans in an iterator-based query processor. See [Gra90, Gra93] for details of the iterator model and the mentioned extensions.

3.2 Query Evaluation Plans

The starting point for a distributed query evaluation is a plan, which looks like a plan for local query processing. The only difference is the existence of three additional annotations for each iterator in this plan. One annotation denotes the execution server for this iterator, another indicates if this iterator (together with the whole subtree rooted at this iterator) should be executed in a new thread. The last additional annotation is a URL, which points to the code for that iterator.

The client machine then starts with a depth-first traversal of the plan in order to instantiate the iterators. If we find an iterator with a site annotation different from the local machine, the iterator together with the subtree rooted at it is sent to the specified site and the traversal starts anew for that subtree. The communication link between the two parts of the distributed plan is provided by the runtime system.

The actions for the threading annotations look similar but are handled with a higher priority during plan distribution than the ones for site annotation. The execution of iterator subtrees in separate threads can be used for exploiting a multi-processor machine, but its main purpose in our system is to reduce the effect of slow communication links.

If the code URL of an iterator points to a WWW-server, the Java class loader local to the query fetches the particular class. Of course predefined iterators can be loaded from the local code base of the server and need not be requested over the network.

3.3 Generating Query Evaluation Plans

In order to test various plans without the interaction of an optimizer, there exists the possibility to specify a query evaluation plan in the form of an XML document. This way is also suitable for composing plans with new operators embedded. For the creation of such a document an XML editor could be used. The XML document can be further enriched with information about parameters needed for the execution of a plan, just like host variables in embedded SQL. We have built a servlet, which uses this information to create an HTML form and presents this form to the user. After the user has specified the values for the parameters, the servlet instantiates the query evaluation plan, initiates the execution of the query and transforms the query result into an HTML page.

The optimization of queries in the context of our query processor is an ongoing work. Currently we have an optimizer capable of generating plans in the way the optimizer used in the Garlic project [HKWY97] works. But as stated earlier our system is more flexible than the Garlic approach, which means that optimization will be more difficult.

3.4 Security

The security problem we are talking about in this section goes beyond authorization or authentication normally addressed in discussions about this subject in database systems. Obviously, a distributed system like ours must support authorization and authentication like any other database. In addition, special security issues arise from the extensibility. Code supplied by a user can be executed at any server-site. Without special security precautions nobody would be willing to run a copy of our query processor on their sites. The imported code has to be executed in a "sandbox" protecting the server machine from hostile users. Fortunately, Java 2 provides a rich framework for solving such security issues, which we used in our implementation.

4 Conclusion

We have installed our system on several sites at the University of Passau and, for testing in a larger scale environment, at the University of Mannheim and the University of Maryland, CP. We demonstrate how it works with a small "demo" travel agency application as described in Section 2.

References

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, March/April 1997.

[Gra90] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 102–111, Atlantic City, NJ, USA, June 1990.

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

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

[MS97] M. Tork Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *VLDB'97, Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.