HyperQueries: Dynamic Distributed Query Processing on the Internet*

Alfons Kemper Christian Wiesner

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

Abstract

In this paper we propose a new framework for dynamic distributed query processing based on so-called *HyperQueries* which are essentially query evaluation sub-plans "sitting behind" hyperlinks. We illustrate the flexibility of this distributed query processing architecture in the context of B2B electronic market places. Architecting an electronic market place as a data warehouse by integrating *all* the data from *all* participating enterprises in one centralized repository incurs severe problems. Using HyperQueries, application integration is achieved via dynamic distributed query evaluation plans. The electronic market place serves as an intermediary between clients and providers executing their sub-queries referenced via hyperlinks. The hyperlinks are embedded within data objects of the intermediary's database. Retrieving such a virtual object will automatically initiate the execution of the referenced HyperQuery in order to materialize the entire object. Thus, sensitive data remains under the full control of the data providers.

1 Introduction

Electronic market places and virtual enterprises have become very important applications for query processing [Jhi00]. Building a scalable virtual B2B market place with hundreds or thousands of participating suppliers requires highly flexible, distributed query processing capabilities. Architecting such an electronic market place as a data warehouse by integrating *all* the data from *all* participating enterprises in one centralized data repository incurs severe problems:

Security and privacy violations: The participants of the market place have to relinquish the control over their data and entrust sensitive information (e.g., pricing conditions) to the market place host.

Coherence problems: The coherence of highly dynamic data, such as availability and shipping information, may be violated due to outdated materialized data in the market place's data warehouse.

Schema integration problems: Using the warehouse approach all relevant data from all participants have to be converted à priori into the same format. Often, it would be easier to leave the data inside the participant's information systems, e.g., legacy systems, within the local sites, and apply local wrapper/transformer operations. This way, data is only converted *on demand* and the most recent coherent state of the data is returned.

Fixed query operators: In a fully integrated (data warehouse-like) electronic market place, all information is converted into materialized data. This is often not desirable in such complex applications like electronic procurement/bidding. For example, in pricing offers one would like to have vastly different choices:

- fixed pricing via materialized data
- operators which calculate the prices based on a multitude of local and global parameters (identity of the consumer company, availability, local plant utilization, sub-contractor prices, etc.)
- even human interaction during the processing of such complex e-procurement queries is desirable.

We propose so-called *HyperQueries* to architect highly flexible distributed query processing systems. HyperQueries are essentially query evaluation sub-plans "sitting behind" hyperlinks. This way the electronic market place can be built as an intermediary between the client (issuing a query) and the providers executing their sub-queries referenced via hyperlinks. The hyperlinks are

*This work was supported by the German National Research Council (DFG) under Contract Ke 401/7-1

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

embedded as attribute values within data objects of the intermediary's database. Retrieving such a virtual object automatically initiates the execution of the referenced HyperQuery in order to materialize the entire object. Thus, sensitive data can remain under the full control of the data providers. Instead of replicating the data at the intermediary, only the hyperlink is embedded.

In summary, the HyperQuery framework allows to blur the distinction between the allocation schema and the data—as it is found in clear separation in traditional distributed databases. In our prototype implementation, called QueryFlow,[1] we distinguish between hierarchical and broadcast processing of HyperQueries. In the hierarchical processing mode the initiator of a HyperQuery is in charge of collecting the processed data. Under broadcast processing the data objects containing hyperlinks are sent to corresponding HyperQueries which will then be in charge of routing the processed objects to the query initiator or to further HyperQueries, if the objects contain additional hyperlinks.

1.1 Related Work

Distributed databases have been studied since the late seventies [WDH+81, Sto85]. Middleware systems [TRV96, HKWY97] try to overcome the heterogeneity faced when data is dispersed across different data sources. [SL90] discusses reference architectures for federated DBMSs from system and schema viewpoints. Our distributed query processor ObjectGlobe [BKK+01] integrates dispersed data sources and provides the dynamic loading of functionality from external code repositories. Cohera [HSC99], based on the economic model of Mariposa [SAL+96], integrates heterogeneous databases using replication tools. Continuous queries in NiagaraCQ [CDTW00] allow users to receive new results when they become available. In [MMM97] WebSQL is used to "query the Web" in navigational style. [GW00] combines the query facilities of traditional databases with existing search engines on the Internet. [LSK95] queries a central mapping information of all participating, distributed data sources.

Stonebraker et.al. [SAHR84] propose a related approach to our HyperQueries. But their work is restricted to stored queries in centralized databases. SOAP [BEK+00] provides a mechanism for exchanging information between distributed applications using XML.

[YP00] describes a reference architecture for interoperable e-commerce applications. Virtual enterprises and B2B e-commerce environments present an important application domain for our new technique: the automobile industry's electronic market place endeavor "Covisint" [Cov] and SAP's "mySAP.com" [SAP99] market places are among the well known examples.

[1] The name of our system was derived from *query* processing and work*flow* systems because processing queries with HyperQueries bears some similarities with processing distributed workflows by routing documents to the appropriate tasks.

```
select p.ProductDescription, c.Supplier, c.Price
from NeededProducts p, Catalog@MarketPlace c
where p.ProductDescription = c.ProductDescription
order by p.ProductDescription, c.Price
expires Friday, May 18, 2001 5:00:00 PM CET
```

Figure 1: Example Query of the Car Manufacturer

1.2 Running Example

We demonstrate the HyperQuery technique with a scenario of the car manufacturing industry. We assume a hierarchical supply chain of suppliers and subcontractors. A typical process of e-procurement to cover unscheduled demands of the production is to query a market place for these products and to select the incoming offers by price, terms of delivery, etc. The price of the needed products can vary by customer/supplier-specific sales discounts, duties, plant utilization, etc.

In traditional distributed query processing systems such a query can only be executed if a global schema exists or all databases are replicated at the market place. Considering an environment, where hundreds of suppliers participate in a market place, one global query which integrates the sub-queries for all participants would be too complex and error-prone.

Following our approach the suppliers have to register their products at the market place, which they want to participate in, and specify, by which sub-plans the price information can be computed at *their* sites. This calculation can be arbitrarily complex and involve their sub-contractors, too. The allocation schema given by the data at the market place is exploited for execution.

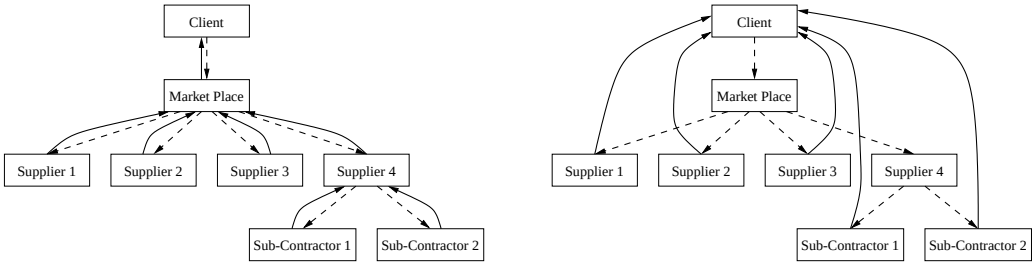
Figure 1 shows an SQL-like query, that returns the prices and suppliers of all needed products. Figure 2 shows two possible execution traces of this query—both are supported by our evaluation technique. In the hierarchical execution of Figure 2(a) the resulting objects flow back to the sites, where the original input objects came from, whereas in the broadcast execution of Figure 2(b) the objects do not flow all the way through intermediates back to the client, but are routed directly to the client.

1.3 Overview

The rest of this paper is organized as follows. Section 2 defines HyperQueries. Section 3 illustrates the execution of HyperQueries in our distributed query processor and shows different kinds of query operators in subplans. In Section 4 we develop some optimization techniques for HyperQuery execution and discuss some details of our implementation. We show first experimental performance results of our approach in Section 5 and conclude in Section 6.

2 HyperQueries: Syntax and Semantics

Our approach is based on virtual attributes whose values are determined by evaluating remote sub-queries



(a) Hierarchical Query Processing (b) Broadcast Query Processing

Figure 2: Flow of Control and Flow of Objects
(Dashed lines indicate the flow of control and intermediate results, solid lines indicate the flow of result objects)

on demand. In contrast to [SAHR84] we do not store the queries, i.e., the sub-plans to be executed at data providers, within virtual attributes. Instead, hyperlinks referencing the sub-plans are embedded as virtual attributes within the data. The queries themselves are located at the distributed data providers (remote hosts). We refer to hosts that manage data with virtual attributes as intermediaries.

In the following we show how hyperlinks and HyperQueries can be incorporated into the database design. We chose a relational schema for the market place and XML as data model for all data being exchanged by HyperQueries, as XML is the emerging standard for data exchange and relational and object-relational data sources can easily be integrated.

2.1 Metadata for HyperQuery Processing

We introduce *virtual attributes* to encapsulate hyperlinks and the results of the corresponding sub-plans as columns in a database table. The hyperlinks are replaced by the result values of the sub-plans, whereby a certain schema, that is defined and publicly available at the intermediary, has to be obeyed. This schema covers the public intermediary's tables and the values that are calculated by the HyperQueries when following the hyperlinks. It further defines application specific parameters, e.g., `Quantity`, that can be used by remote hosts to calculate the actual value of virtual attributes.

2.2 Specification of Hyperlinks

We specify hyperlinks by defining a Uniform Resource Identifier (URI) schema (see Figure 3). The individual components of these URIs have the following meaning: the leading `hq` denotes our *HyperQuery* protocol, `<HostDNS>` is the DNS name of the host, on which the sub-plan is executed, and `<PathToPlanId>` is the name of the stored sub-plan within a repository of plans. The `<HostDNS>` and the `<PathToPlanId>` are referred to as URI prefix. Both the optional global parameter list and the object specific parameters are "&"-separated key-value lists. The former parameterize remote sub-plans, e.g., `Currency` for payment. The latter represent foreign keys on a virtual table at the remote host and have to contain enough information, to calculate the actual value of the virtual attribute at the remote site. All entries of the virtual attribute with the same URI prefix must have the same parameter structure.

Figure 4 shows a simple extension of the `Catalog` table of the electronic market place example of Section 1.2. The virtual attribute `Price` of the first tuple denotes, that the price is calculated at the host `supplier1.com` for an object with the key attribute `ProdID` and value `CB1232` by using the sub-plan named `Electrical/Price`. The fourth tuple requires the additional global parameter `Currency`.

2.3 Syntax of HyperQueries

A HyperQuery is the counterpart of the hyperlinks of virtual attributes. These query plans are executed on remote hosts and may be arbitrarily complex, integrate applications, ERP- and legacy systems, and may even comprise user interaction. The most comfortable way for stating HyperQueries is to use our SQL dialect. A HyperQuery using SQL accesses a virtual table called `HyperQueryInputStream` that serves as a receiver of the input data objects that "flow through" the hyperlinks. The schema of this virtual table is composed of all object specific parameters of the corresponding hyperlink and application specific parameters that are transmitted during hyperlink processing but not contained within the hyperlink. Additional attributes of an input data object are not accessible within the HyperQuery; they are passed through. [RS97] describes how more complex data sources can be queried using SQL by defining views over legacy systems. In our QueryFlow system, alternatively, a HyperQuery could consist of arbitrarily complex Java operations which have to implement the iterator interface of [Gra93] (cf. Section 3.5).

Figure 5 shows the SQL formulations of two example HyperQueries, both determining the price of the specified products. Note that `p.Price` and `p.ComponentPrice` in both HyperQueries can be virtual attributes, i.e., further HyperQueries on other hosts could be executed to compute the value.

```
<hqschema>  ::= "hq://"<HostDNS>"/"<PathToPlanId> ["!"<GlobPL>] "?"<ObjPL>
<GlobPL>    ::= <GlobPN>"="<GlobPVal> {"&"<GlobPN>"="<GlobalPVal>}
<ObjPL>     ::= <ObjPN>=<ObjPVal> {"&"<ObjPN>"="<ObjPVal>}
```

Figure 3: URI Schema of the *HyperQuery* Protocol

ProductDescription	Supplier	Price
Battery, 12V 32 A	Supplier 1	`hq://supplier1.com/Electrical/Price?ProdID=CB1232`
Battery, 12V 55 A	Supplier 1	`hq://supplier1.com/Electrical/Price?ProdID=CB1255`
Tires 175/65TR14	Supplier 2	`hq://supplier2.com/Price?ProdKey=175/65TR14`
Spark Plug VX	Supplier 3	`hq://supplier3.com/PriceForUSA!Currency=USD?ID=1234`
Alternator 50 A	Supplier 4	`hq://supplier4.com/RegularPrice?SerialNo=Alt50`
...	...	...

Figure 4: Sample Extension of the `Catalog@MarketPlace` Table

```
select h.*, p.Price as Price
from HyperQueryInputStream h, Products p
where h.ProdID = p.ProdID
```

(a) `Electrical/Price` at Supplier 1

```
select h.*, (h.Quantity * sum(p.ComponentPrice)) as Price
from HyperQueryInputStream h, BillOfMaterial b, Parts p
where h.SerialNo = b.SerialNo and b.ComposedOf = p.PartID
group by h.*
```

(b) `RegularPrice` at Supplier 4

Figure 5: Two Example HyperQueries in Our SQL Dialect

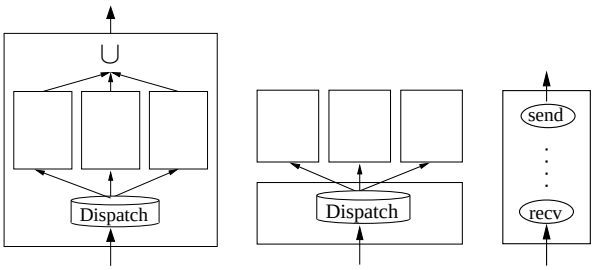
2.4 Interface to HyperQueries

If an object is sent to a HyperQuery, the URI of the virtual attribute is replaced by the actual value. This value is calculated from the object specific and the application specific parameters. The former are given by the URI, the latter stem from the globally available schema definition at the intermediary. Other attributes of the object cannot be used for the computation within the HyperQuery and are passed through.

The type of the actual value of the virtual attribute has to match the schema definition given at the intermediary; objects of incompatible type are discarded. If the type is single-valued and multiple values for the virtual attribute are computed, multiple objects have to be returned.

3 HyperQuery Execution in our QueryFlow System

In this section we illustrate the execution of HyperQueries in our QueryFlow system. The QueryFlow system is a distributed and open query processor for data sources on the Internet. The whole system is written in Java for two reasons: First, Java is portable, so that our system can be installed with very little effort: hosts need to install the QueryFlow system and can then very easily join a market place by inserting hyperlinks at the intermediary and providing the corresponding sub-plans. Second, Java provides secure extensibility. Like the QueryFlow system itself, user-defined query operators are written in Java. They could be loaded from remote sites (e.g., the market place host or third-party vendors) on demand. For security reasons they are executed in their own Java "sandbox".



(a) Nesting (b) Sequencing (c) Inner

Figure 6: The Three Possible Templates for Sub-Plans

3.1 Templates for Sub-Plans

Users of the QueryFlow system specify HyperQueries using our SQL dialect as described in Section 2.3. A query is transformed into an operator tree that is stored as a sub-plan in a local repository. The three possible templates for sub-plans in our QueryFlow system are illustrated in Figure 6 and can be characterized as follows:

Nesting Sub-Plans: As shown in Figure 6(a) these sub-plans contain a `Dispatch` operator that splits one input stream into multiple output streams that serve as input streams for the nested sub-plans. This operator is the basic operator for processing HyperQueries (cf. Section 3.2 and Section 4). The `Union` (re-)merges the output streams of the nested sub-plans and produces one output stream. Thus, the flow of objects is totally encapsulated inside a sub-plan of this pattern. The client query is always transformed into a plan of this kind.

Sequencing Sub-Plans: Sequencing sub-plans as shown in Figure 6(b) contain the initial `Dispatch` operator that splits one input stream into multiple output streams; no final `Union` is given, so a surrounding sub-

ProductDescription	Quantity
Battery, 12V 32 A	500
Battery, 12V 55 A	750
Tires 175/65TR14	1000
Spark Plug VX	8000

Figure 7: Needed Products of the Car Manufacturer

plan with a `Union` is required to which the objects of the sub-plans can be routed. Thus, objects that are once sent to the next sub-plan are never sent back to the delegating sub-plan. So the further processing of the data objects is beyond the control of the initiating sub-plans.

Inner Sub-Plans: Figure 6(c) shows sub-plans that have one input stream and one output stream. These sub-plans form the innermost parts of the query execution where the actual values of virtual attributes are determined. As already mentioned above multiple output objects may be generated for one input object.

3.2 Processing Hyperlinks

In our QueryFlow system processing hyperlinks is done by an operator, called `Dispatch`, that splits one input stream into multiple output streams. If a hyperlink is encountered, the actual value is computed by "following" the hyperlink according to this procedure:

1. The hyperlink is split into its components, i.e., the DNS of the remote host, the identifier for the sub-plan, the global parameters, and the object specific parameters. The object specific parameters are merged with the current input object.

2. If the referenced sub-plan has not yet been instantiated at the remote host, an instantiation request containing the global parameters is sent.

3. Once the sub-plan has been instantiated, all objects with the same URI prefix are routed to it, whereby whole objects as produced by step 1 are sent.

3.3 Processing a Simple Query

On the basis of our running example we illustrate the process of incremental plan generation and plan execution. Figure 7 shows the `NeededProducts` table.

Due to clarity we substituted in Figure 8 the concrete data objects by symbols, where □ and △ denote the two battery objects, ○ denotes the tires object, and ◇ denotes the spark plug object. Figure 8(a) shows the start of the query execution: The user-stated plan is instantiated with a scan of the `NeededProducts` table at the client. The attributes `Price` and `Supplier` of the `Catalog` table at the market place are joined (indicated by ⋈⁓) with the input objects. The vertical hatch indicates the enriched objects in the following figures.

As `Price` is a virtual attribute, a `Dispatch` operator splits the resulting stream of objects is split into multiple output streams. In Figure 8(b) the first object for Supplier 1 passes the `Dispatch` operator which sends an instantiation request[2] for the sub-plan `Electrical/Price` to Supplier 1. Basically this sub-plan consists of a join with a local table as shown in the HyperQuery of Figure 5(a). All objects that belong to this sub-plan are routed to it by the `Dispatch` operator (Figure 8(c)/(d)). Figure 8(d) also shows the processing of the ◍ object at the market place. As its pricing information is calculated at Supplier 2, the instantiation of the corresponding sub-plan is requested. This sub-plan involves a complex application to calculate the price of the input objects. Concurrently the price has been added to the ▥ object at Supplier 1 and the resulting ■ object[3] can be forwarded to the final `Union`. So an additional input stream is requested at the `Union`.

Having registered the new input stream at the `Union`, the ■ object is sent to the market place. The price is inserted into the next data object ◮ and generated a ▲ object (Figure 8(e)). The ◍ object is routed to Supplier 2. The market place requests the instantiation of the sub-plan named `PriceForUSA` for the last input object ⧫ at Supplier 3 and sets the global parameter `Currency` to `USD`. In this sub-plan a human user enters the pricing information, e.g., using a GUI. In Figure 8(f) the ⧫ object is routed to Supplier 3. Supplier 2 has inserted the pricing information into its ◍ object and generated a ● object, which is sent to its target. So a further input stream is requested at the `Union`. Supplier 1 routes its ▲ object to the `Union`.

Supplier 2 sent its ● object to the `Union` and Supplier 3 has inserted the pricing information for its ⧫ object and generated a ◆ object (Figure 8(g)). Figure 8(h) depicts the result, where the actual value of all input objects has been inserted and the resulting objects have reached the `Union`. Based on these data objects the query is processed further, i.e., the sorting is done.

3.4 Processing Complex Queries

So far we have demonstrated the incremental instantiation of sub-plans for simple one-level HyperQuery processing. The HyperQuery concept is, of course, not restricted to one level. While processing a HyperQuery, other hyperlinks may be encountered which initiate nested HyperQueries. Figure 9 illustrates our component-based QueryFlow system on more complex example applications. We only show the complete query plans (after all sub-plans have been instantiated) and omit both concrete data objects and the sequences of the stepwise instantiation of the sub-plans.

3.4.1 Hierarchical HyperQuery Execution

If a remote host encounters a virtual attribute that is needed for the further execution of the HyperQuery,

[2] Note, that all sub-plans are instantiated only once for a query.

[3] Objects with fully materialized virtual attributes are visualized in solid black.

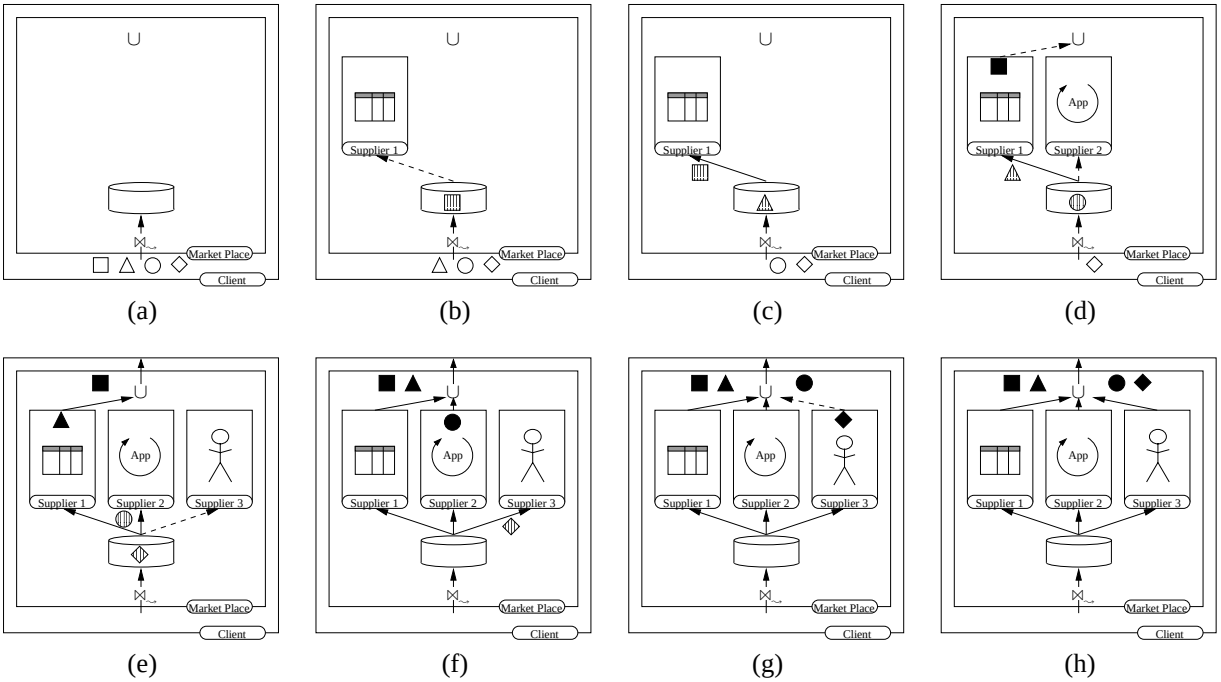


Figure 8: Routing of Objects & Instantiation of Sub-Plans
(solid lines indicate the routing of objects, dashed lines indicate the instantiation of sub-plans)

the host acts as intermediary and initiates a nested sub-query at another remote host using the pattern of Figure 6(a). After pre-processing the data objects, they flow from the surrounding sub-plan to the nested sub-plans, where the value of the virtual attribute is computed. Then the complete objects are sent back to the surrounding sub-plan, where they are processed further. Figure 9(a) shows an example. Supplier 4 executes the HyperQuery of Figure 5(b) and accesses the virtual attribute `ComponentPrice`. Thus, the right hand sub-plan instantiates further nested sub-plans at sub-contractors. Note, that the virtual attributes at the levels of the nesting need not be the same, e.g., the outer virtual attribute could be `Price`, while the inner is `ComponentPrice`.

3.4.2 Broadcast HyperQuery Execution

If a hyperlink is encountered within a HyperQuery and the resulting objects need not be processed any further, the evaluation can be delegated to other HyperQueries. Using sub-plans of the pattern of Figure 6(b) data objects are (after a pre-processing step) forwarded to the sequencing sub-plans. It is the task of the further sub-plans to determine the value of the virtual attribute and to send the resulting objects back to the initiator of the query. The prerequisite is that the virtual attributes are the same for both levels of HyperQuery execution. The `Union` of the surrounding sub-plan merges the results of the broadcast-like inner sub-plans. Figure 9(b) shows an example for the broadcast execution, where Supplier 4 has two subsidiary companies, each one specialized at some goods, and forwards the received objects to them, without post-processing the results.

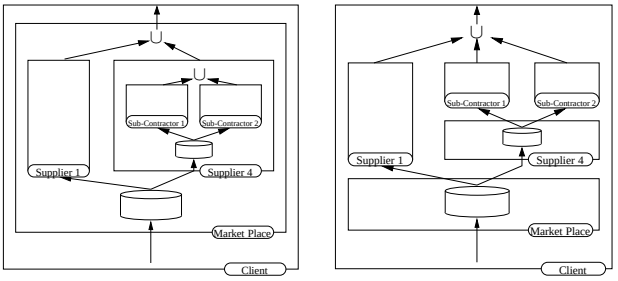


Figure 9: Kinds of HyperQuery Execution

The main advantage of broadcast processing is the quick forwarding of data objects without the need of handling them again at the delegating site. The trade-offs are (1) that the virtual attributes must coincide in all sequencing sub-plans and (2) that many connections, have to register at the merging `Union`. We want to emphasize that it is a local decision of each participating site, what kind of sub-plan is executed, and this decision is not affected by other sites. Thus, it is possible to have both hierarchical and broadcast execution of HyperQueries within the execution of one query.

3.5 Query Operators

As mentioned before, our QueryFlow system provides extensibility. This capability is important, as each participating site has several alternatives for implement-

ing the HyperQueries. So query plans can perfectly be adapted to the companies' local systems. The query plans may contain different kinds of operators which can be characterized by the origin of data.

SQL Database Queries: The simplest kind of local sub-plans are SQL-like queries as shown in Section 2.3. The queries are transformed into a tree containing physical operations of the relational algebra with traditional database operators, e.g., joins, selections, projections, and sorting. Dynamic loading of operators in our QueryFlow system enables the administrator of the local host to integrate new and more efficient database operations into the query execution. One example of such a new database operation is a wrapper that accesses a relational database system using JDBC.

Applications: If complex business applications, e.g., ERP systems like SAP R/3, etc., or legacy systems have to be accessed, wrappers for these applications have to be integrated into the query plan. This is done the same way as database systems are integrated. All we require, is that the wrappers obey the iterator interface. The connection of the QueryFlow system to legacy systems by wrappers means that data is only converted on demand and the most coherent state of the data is returned.

Human Interaction: Determining the values of virtual attributes in sub-plans can even be done by human interaction. In this case a user enters the value of a virtual attribute through a Java applet or a GUI. As these operators are executed at the sites of the owner of the data, sensitive data remains under their full control. These operators have two main parts: a server part, which implements the iterator interface, is specified in the query execution plan, and runs as a part of the query execution. The corresponding client part acts as an input interface.

3.6 Security Issues

Safety is one of the crucial issues in an open and distributed query processing system. Our QueryFlow system provides a security system for authentication, i.e., verifying the identity of a user, authorization, i.e., verifying, if a user has the permission to execute a sub-plan or an operation, and privacy, i.e., denying unauthorized sites access to sensitive data. We extended standard methods to fit the needs of multi-level HyperQuery processing technique. Due to a lack of space we omit the details that can be found in [KW01].

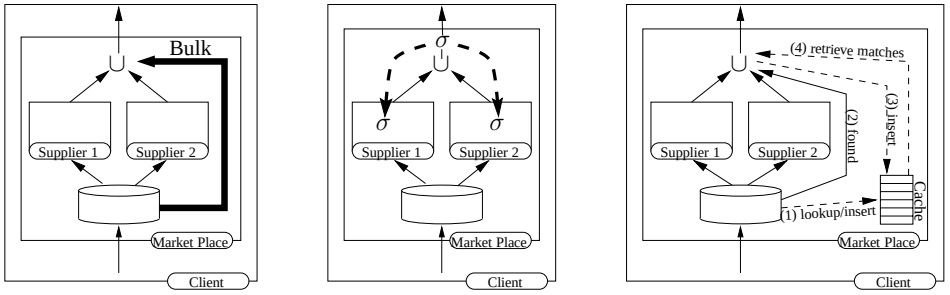
4 Optimization and Implementation Details

So far we have demonstrated the basic techniques for the evaluation of HyperQueries. Now we discuss some optimization approaches and implementation details.

Bypassing of Attributes: If "bulky" attributes, such as images or product descriptions, are requested in the result, that are not needed for the calculation of the virtual attribute at a remote host, they can be projected out when passing the `Dispatch` operator. These attributes are sent to the final `Union` and are re-merged to the resulting objects after the virtual attribute has been calculated. Especially in multi-level HyperQuery execution this decreases the amount of data transferred over the network and reduces the execution time in slow and bursty networks. During the stripping off the bulky attributes a unique sequence number is added both to the bulk objects and the remaining data objects. Using these sequence numbers the bulk objects can be joined to the corresponding data objects. This optimization method is similar to bulk bypassing ([BCKK00, CKKW00]) in central databases. Figure 10(a) illustrates the bypassing of bulk objects around the sub-plans in the market place scenario.

Predicate Migration: Predicates on virtual attributes of the user-stated query cannot be evaluated before the actual value has been computed. To reduce the amount of transferred data, these predicates can be pushed from the user-stated query "into" the HyperQueries at the remote hosts. Thus, only relevant data objects are returned. The implementation of this optimization is straightforward: The selection predicate is sent to the remote site during the instantiation of the sub-plan. When passing objects through the `Send` operator, the selection is performed. Additional profit can be drawn, if the remote hosts incorporate the possibility of predicate migration into their HyperQueries, e.g., the remote hosts can place a `Selection` operator into their sub-plans, whose predicate is set during the instantiation. Figure 10(b) illustrates the migration of predicates.

Multiple Virtual Attributes: If a query requests multiple virtual attributes the naïve execution strategy would request at first the value of the first virtual attribute, then that of the second virtual attribute, etc. If all virtual attributes of an object are evaluated at the same site, the requests can easily be bundled. A plan is generated that contains one `Dispatch` operator for all virtual attributes whose evaluation can be combined. During the execution the `Dispatch` operator sends the list of all requested virtual attributes with the instantiation request for one remote sub-plan. When an object passes the `Dispatch` operator, it is routed to the sub-plan, where the actual values of all virtual attributes are determined at once. This avoids sending one object multiple times to the same host. If not all virtual attributes can be evaluated at the same site, e.g., if the price and the rating by an independent organization are requested, the calculation can be parallelized as follows: The `Dispatch` operator sends *one* input object with a unique sequence number to *all* its corresponding sub-plans. The `Union` re-merges the resulting data objects of different sub-plans using the sequence number. Objects are passed to the next operator, when all virtual attributes have been inserted by the `Union`.



(a) Bulk Bypassing (b) Predicate Migration (c) Intra-Query Caching

Figure 10: Illustrating Optimization Techniques

Caching of Results: Due to duplicates (which may be produced by preceding joins) the same virtual attributes have to be evaluated multiple times. This can be avoided by caching. The evaluation of virtual attributes is similar to the invocation of expensive methods, but in contrast to [HN96] this is done asynchronously, i.e., objects are sent to sub-plans, before the results of previous objects are returned.[4] Thus it is not sufficient to store only the returned values. We also have to keep book of objects that were sent to sub-plans and have not yet produced a result. Figure 10(c) depicts the hash table based caching of virtual attributes. On any input object the `Dispatch` operator probes the hash table (1). A cache hit is directly sent to the `Union`, bypassing the HyperQueries (2). Otherwise the object is inserted into the hash table as a request. If it was the first request for this URI, the object is sent to the corresponding HyperQuery. If a result from a HyperQuery is received by the `Union`, it is inserted into the hash table (3) and the pending objects with the same URI are returned (4). If assuming that the results are highly dynamic and for coherence reasons cannot be re-used in another query, the hash table has to be discarded when the query execution has finished. But if the remote hosts give runs of validity for their results, this approach can be extended to inter-query caching, where results are cached until expiry.

Implementation of the Dispatch Operator: The most significant operator for processing HyperQueries is the `Dispatch` operator which splits the input stream into multiple output streams. As the number of resulting output streams cannot be determined á priori, we have to fork one `Dispatch` operator for each new output stream. The first instantiated `Dispatch` operator also acts as the coordinator for the other forked operators and keeps book of them. Each `Dispatch` operator runs in a separate thread; all `Dispatch` operators share one common input stream, from which each `Dispatch` operator selects its relevant objects. Thus we obtain the concurrent and independent routing of objects to the sub-plans on all participating hosts.

[4]For the same reason sorting on the URI does not work well in HyperQuery processing, as this would be the worst case for the asynchronous approach.

Support of Long-Running Queries: To limit the duration of queries, we introduced the `expires`-clause, which allows us to give a time-to-live (TTL) for queries. Each instantiated sub-plan is annotated with this TTL and is monitored on expiry. If the TTL elapses, the sub-plans are aborted and only the objects gathered so far are considered for the (approximate) result. As the TTL can amount to days and open network connections are error-prone, connections that have not transferred data for a certain time but are still active, are closed temporarily, without affecting the query execution itself.

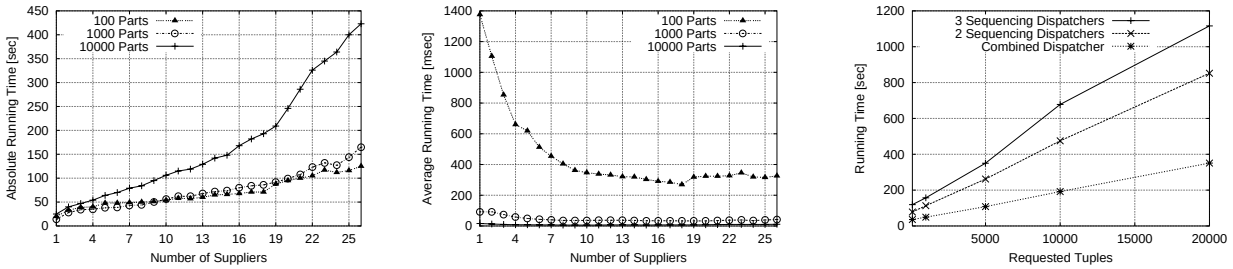
Fault Tolerance: If a host does not respond to any request, either a network failure occurred or the host is down. As this could only be a short-term break and the processing of a query lasts longer, we store all objects that belong to the failure host and periodically re-try to connect to the host. If the remaining query has finished before the host responds, it is no longer waited for, and the client is informed about the incomplete result. If the host is accessible again while the remaining query runs, the sub-plan is instantiated at the host, and the stored objects are sent to the host where query processing continues as regular.

5 Performance Investigation

In this section we present a few initial benchmark results obtained from our QueryFlow system. In particular, we concentrate on investigating the scalability of our approach in a distributed environment and show the effectiveness of the combination of multiple `Dispatch` operators.

5.1 Experimental Environment

Our test scenario constitutes a market place with 26 suppliers. The data for our databases was taken from the TPC-D [TPC99] benchmark suite of scale factor 1.0. To suit our limited benchmark environment, we converted the 10000 suppliers round robin by SUPPKEY to only 26 suppliers, each being allocated to an individual host. The PARTSUPP table represented the market place and the PART table was partitioned horizontally to obtain several $\mathsf{PART@SUPP}_i$ tables that contained those parts that the supplier i produced. Thus, each supplier offered



Varying the Number of Suppliers — Combining Multiple `Dispatch` Operators

(a) Absolute Running Times [sec] (b) Average Running Times [msec] (c) Absolute Running Times [sec]

Figure 11: Running Times

approximately 30000 parts, whereby each part was produced by 4 suppliers which lead to 800000 entries at the market place and 200000 distinct parts. SUPPLYCOST and AVAILQTY became virtual attributes. The databases were stored in commercial relational database systems. Each participant of the market place ran its database server on a separate host, whereby the market place was placed on a Sun Enterprise 450 with four 400 MHz UltraSparc II processors and 4 GByte memory. The other database servers ran on machines of type Sun Ultra 10 with 1 UltraSparc IIi processor at 333 MHz and 128 MByte memory. All hosts were in the same 100 MBit LAN, running Solaris 2.7 and using Sun's JDK 1.2.2 as the basis for our QueryFlow system.

5.2 Scalability of HyperQuery Processing

For the first test we varied the number of requested suppliers in our market place from 1 to 26. The query issued by the client was:

select PARTKEY, COMMENT, SUPPKEY, SUPPLYCOST
from PARTSUPP
where PARTKEY <'[sel]' **and** SUPPKEY < '[supps ']

We used '[supps]' to limit the number of requested suppliers. Further we varied '[sel]' and requested 100, 1000 and 10000 parts. As the parts were distributed among 26 suppliers and each part was offered by 4 suppliers, $4 \cdot sel \cdot supps/26$ objects were returned. The following HyperQuery was invoked at Supplier i:

select h.*, RETAILPRICE
from PART@SUPP$_i$, HyperQueryInputStream h
where PARTKEY = h.PARTKEY

As the number of resulting objects varied with the number of requested suppliers, we normalized the results to the average time. The absolute running times for this experiment, shown in Figure 11(a), are as expected: the more suppliers take part, the bigger the market place is, and the longer a query runs. But Figure 11(b) proves that the average running time per resulting object decreases with a higher number of suppliers. This is an indication that the increase of costs caused by additional registered suppliers is sub-linear because of the parallel HyperQuery processing. Comparing the average times of 100 parts and 10000 parts of Figure 11(b) shows that with an increasing number of requested objects the average time per object decreases by orders of magnitudes. This results from the fact that the fixed costs of the plan instantiation and the authentication overhead amortize with increasing number of processed objects.

5.3 Evaluating Multiple Virtual Attributes

In this experiment we demonstrate the benefits of bundling requests for multiple virtual attributes. All 26 suppliers were incorporated in this experiment. The first query accessing the two virtual attributes SUPPLYCOST and AVAILQTY was:

select PARTKEY, SUPPKEY, COMMENT, '[virtual_attrs]'
from PARTSUPP
where PARTKEY <'[sel]'

We varied the selectivity '[sel]' of the query, i.e., the number of requested data objects from 100 up to 20000. The naïve plan had two sequencing `Dispatch` operators requesting at first SUPPLYCOST and then AVAILQTY, the optimized variant combines the evaluation of both virtual attributes in one request. For a second query we extended the schema of the PARTSUPP table by adding an additional virtual attribute SHIPCOST and queried three virtual attributes SUPPLYCOST, AVAILQTY, and SHIPCOST.

Figure 11(c) shows the running times for the queries. The optimized variants are about a factor of 2 (3) faster than the naïve plans with two (three) sequencing `Dispatch` operators, as the objects are sent only once. Figure 11(c) shows only one plot of the optimized query, as we found out, that it is neglectable, if the HyperQuery joins two or three attributes to the input objects.

6 Conclusions

In this paper we proposed a framework for dynamic distributed query processing based on HyperQueries which are essentially query evaluation plans "sitting behind" hyperlinks. We illustrated the flexibility of this distributed query processing architecture in the context of B2B electronic market places. Architecting an electronic market place as a data warehouse by integrat-

ing all the data from all participants in one centralized repository incurs severe problems. Using HyperQueries, application integration is achieved via distributed query evaluation plans. Now the electronic market place serves as an intermediary between clients and providers executing their sub-queries referenced by hyperlinks. We demonstrated how these hyperlinks can be embedded in the intermediary's database as so-called virtual attributes. Further we illustrated the execution of HyperQueries and pointed out more complex scenarios that can be divided into hierarchical and broadcast HyperQuery execution. Further we demonstrated some effective optimization techniques for HyperQuery processing, described some important implementation details and investigated the scalability of our technique.

Acknowledgments

We would like to thank the anonymous reviewers and the ObjectGlobe team for their helpful comments.

References

[BCKK00] R. Braumandl, J. Claussen, A. Kemper, and D. Kossmann. Functional join processing. *The VLDB Journal*, 8(3-4):156–177, 2000.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP`, May 2000.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal*, 2001 (to appear in "Special Issue on E-Services").

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 379–390, June 2000.

[CKKW00] J. Claussen, A. Kemper, D. Kossmann, and C. Wiesner. Exploiting early sorting and early partitioning for decision support query processing. *The VLDB Journal*, 9(3): 190–213, December 2000.

[Cov] Covisint. `http://www.covisint.com/`.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GW00] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 285–296, June 2000.

[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, August 1997.

[HN96] J. Hellerstein and J. Naughton. Query Execution Strategies for Caching Expensive Methods. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 423–434, June 1996.

[HSC99] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, Open Enterprise Data Integration. *IEEE Data Engeneering Bulletin*, 22(1):43–49, March 1999.

[Jhi00] A. Jhingran. Moving up the food chain: Supporting E-Commerce Applications on Databases. *ACM SIGMOD Record*, 29(4):50–54, December 2000.

[KW01] A. Kemper and C. Wiesner. HyperQueries: Dynamic Distributed Query Processing on the Internet. Technical report, Universität Passau, Fakultät für Mathematik und Informatik, October 2001. Available at `http://www.db.fmi.uni-passau.de/publications/papers/HyperQueries.pdf`.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems (JIIS)*, 5(2):121–143, 1995.

[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.

[RS97] M. Tork Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, August 1997.

[SAHR84] M. Stonebraker, E. Anderson, E. Hanson, and B. Rubenstein. QUEL as a Data Type. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 208–214, June 1984.

[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`.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Sto85] M. Stonebraker. The Design and Implementation of Distributed INGRES. Addison-Wesley, Reading, 1985.

[TPC99] Transaction Processing Performance Council TPC. TPC Benchmark D (Decision Support). Standard Specification 2.1, Transaction Processing Performance Council (TPC), February 1999. `http://www.tpc.org/`.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *Proc. of the Intl. Conf. on Distributed Computing Systems*, pages 449–457, 1996.

[WDH+81] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An Overview of the Architecture. IBM Research, RJ3325, December 1981. Reprinted in: M. Stonebraker (ed.), Readings in Database Systems, Morgan Kaufmann Publishers, 1994, pp. 515–536.

[YP00] J. Yang and M. P. Papazoglou. Interoperation Support for Electronic Commerce. *Communications of the ACM*, 43(6):39–47, June 2000.