ubQL, a Language for Programming Distributed Query Systems

(Extended Abstract)

Arnaud Sahuguet Val Tannen

{sahuguet,val}@saul.cis.upenn.edu

University of Pennsylvania

Abstract

ubQL is a distributed query language for programming large-scale distributed query systems such as resource sharing systems. The language is obtained by adding a small set of mobile process primitives (communication channels, migration operators, etc.) on top of any traditional query language. Queries are encapsulated into processes and can migrate between sites thus enabling cooperation. An important methodological device is the separation of the *installation* (including *migration*) of query processes from the distributed *execution* of the queries.

In this paper, we give an overview of ubQL, show how to encode widely used *distributed query patterns* such as chaining, recruiting, query/data/hybrid shipping, etc., and evaluate some language-based rewrite strategies for the installation of ubQL queries that use only partial and distributed knowledge of execution costs.

1 Introduction and Motivation

Our work was motivated by distributed query systems that do not fit well within the traditional distributed databases paradigm [7]. A prime example are systems for the large-scale distribution or exchange of resources such as software packages, scientific data, or multimedia files. There are several sources of complexity in such systems. In addition to sheer scale, there may be many layers of sites between a query site and the resource sites, as well as significant variability. This puts limits on what "central" sites can know and do and also on what individual sites can achieve by looking around aimlessly to discover resources. In such systems, query execution can only be planned in a distributed way using just partial and local knowledge. Techniques such as brokering, proxying, caching, publish/subscribe, advertising, chaining, referral, recruiting, etc., become essential in such systems. Existing architectures select just one or two techniques from this list, hard-code them, and then build around. The resulting architectures scale with difficulty.

To make such systems much more scalable, we propose a *distributed query language*, ubQL; in fact, a set of primitives for process manipulation that can be added to *any* query language. The language encapsulates queries into processes and uses process migration as the basic primitive for cooperation between sites. Thus, instead of hard-coded, we make the techniques employed by various sites *programmable* in ubQL. The software running at each site can be seen as a copy of the interpreter of ubQL. It supports separately query process *installation* and then distributed query *execution*. The language relies on a small set of primitives capturing concurrent interaction, process migration and communication via channels. Our approach is to identify a few basic rewrite steps that each site supports and then show how these plus migration can be used to implement all the techniques mentioned above (we call these "distributed query patterns" in analogy with software design patterns). Other benefits of choosing a small set of basic mechanisms are reliability of the implementation and the ability to reason formally about the behavior of the interpreter.

The rest of the paper is organized as follows. In Section 2, we present ubQL language constructs. We then revisit some traditional distributed query patterns and show they can be encoded in the language. In Section 4, we explain some installation algorithms that use only partial and distributed knowledge and show how they compare to traditional distributed query optimization for a specific experimental configuration. We finally

present our conclusions and some future work.

2 The Language

Language Design
The starting point was the desire to express/describe the following elements:
- data, both virtual (views or aliases) and physical (files, relations)
- queries
- sites (virtual data is defined, physical data is located and queries are evaluated at various sites)
- processes that describe the concurrent execution of multiple queries at various sites
- migration, to permit queries to move from site to site
- channels, to support communication among processes

An important requirement is independence of the *underlying query language* that handles the specific data (eg., SQL, XQuery [9], OQL). Still, the data will consist of large values and *streaming* communication must be allowed by ubQL. Finally, we believe that using a small set of *orthogonal* primitives improves optimization opportunities.

Language Overview
The building bricks of the language are *query processes* (*qp*). The language borrows ideas from mobile process calculi [6] especially the concept of *query process migration*. ubQL consists of **sites**, **channels**, **expressions** and **query processes**.
The underlying query language supplies data-specific operators (eg., select-from-where). ubQL expressions are built from these as well as **file names** (data available locally), **aliases** (view names, defined locally or at remote sites) and local **channel names** (data arriving on the channel).
Processes consist of a left hand side, a right hand side and a state. The right hand side is always an expression. They can be of two kinds, as described below:

Syntax	channel ⇐ exp [state] state ∈ {pending, pending@site, execute}	alias ⇐ exp [state] state ∈ {replicated}
Meaning	*"evaluate* exp *and send it on* channel*"*	*"replace* alias *with* exp*"*

A process is always located at a given site (state pending or execute), but can migrate to another site (state pending@remote). Pending and replicated query processes can interact to produce new query processes. During such an interaction, pending *qp*'s disappear during the interaction (they are consumed) while replicated *qp*'s remain (they are used but a copy remains).

Examples of query processes and expressions are presented in Figure 1.

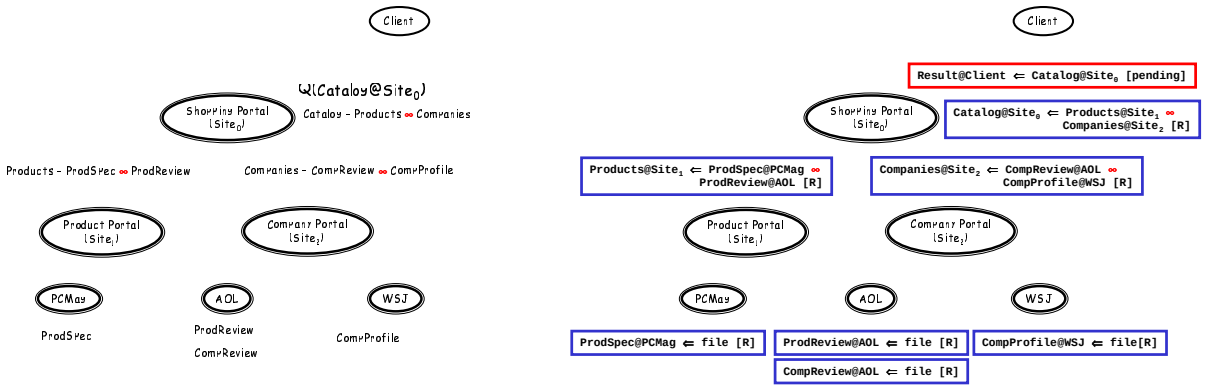

Figure 1: A typical mediator architecture (left) and its encoding in ubQL (right)

The operational semantics of the language consists of two separate stages: **installation** and **execution**. A useful analogy is that installation is like laying down pipes between the site that originated the query and the

sites that have the data, using migration to make use of intermediary sites along the way. Once the pipes are laid down, we can "turn on the faucets" and this is the execution stage. The data flows and it is processed into the eventual answer.

To a large extent, ubQL execution follows established experience in distributed databases. The data is streamed through pipelined operators (one does not need all the data to start computing) using channels between sites. Techniques such as XJoin [8] can be used profitably. The originality of ubQL is its treatment of installation and in the rest of the paper, we will focus only on installation.

Installation
The installation of query processes is based on three simple rewrite rules: **merge**, **split** and **tag**.

• *Merge*
Merge is inspired from the communication reduction rules of π-calculus-like languages [6]. It is used – among other things – to resolve local views. Merge can be applied on two pending *qp*'s or one pending and one replicated *qp*. The merge occurs when the left-hand side of one *qp* (pending or replicated) matches with the right-hand side of a pending *qp*, as illustrated in the table below. Replicated query processes do not get consumed by the rewrite.

$$\begin{array}{l} \mathsf{out} \Leftarrow \mathsf{E}_1(\mathsf{view@local})\ [\mathsf{pending}] \\ \mathsf{view@local} \Leftarrow \mathsf{E}_2\ [\mathsf{pending}] \end{array} \xrightarrow{MERGE} \mathsf{out} \Leftarrow \mathsf{E}_1(\mathsf{E}_2)\ [\mathsf{pending}]$$

$$\begin{array}{l} \mathsf{out} \Leftarrow \mathsf{E}_1(\mathsf{view@local})\ [\mathsf{pending}] \\ \mathsf{view@local} \Leftarrow \mathsf{E}_2\ [\mathsf{replicated}] \end{array} \xrightarrow{MERGE} \begin{array}{l} \mathsf{out} \Leftarrow \mathsf{E}_1(\mathsf{E}_2)\ [\mathsf{pending}] \\ \mathsf{view@local} \Leftarrow \mathsf{E}_2\ [\mathsf{replicated}] \end{array}$$

• *Split*
Split can be seen as the opposite of merge. It is used to separate a query process into *some* sub-query-processes that will perform parts of the computation. It is crucial that the semantics of the original expression be conserved. In the example below, the expression E can be split into $\mathsf{E}_1(\mathsf{E}_2)$ only if we have $\mathsf{E} \equiv \mathsf{E}_1(\mathsf{E}_2)$. Note that this equivalence usually depends on the semantics of the underlying query language.

$$\mathsf{out} \Leftarrow \mathsf{E}\ [\mathsf{pending}] \quad \overset{\text{if } \mathsf{E} \equiv \mathsf{E}_1(\mathsf{E}_2)}{\xrightarrow{SPLIT}} \quad \begin{array}{l} \mathsf{out} \Leftarrow \mathsf{E}_1(\mathsf{ch@local})\ [\mathsf{pending}] \\ \mathsf{ch@local} \Leftarrow \mathsf{E}_2\ [\mathsf{pending}] \end{array}$$

• *Tag*
Tag changes the state of a query process. Query processes that do not need further installation are marked for execution. Query processes that cannot be further installed locally (because they require some interaction with remote *qp*'s) are marked for migration.

$$\mathsf{out} \Leftarrow \mathsf{E}\ [\mathsf{pending}] \quad \overset{\text{if E contains only files and channels}}{\xrightarrow{TAG}} \quad \mathsf{out} \Leftarrow \mathsf{E}\ [\mathsf{execute}]$$

$$\mathsf{out} \Leftarrow \mathsf{E}\ [\mathsf{pending}] \quad \overset{\text{if E contains only remote aliases}}{\xrightarrow{TAG}} \quad \mathsf{out} \Leftarrow \mathsf{E}\ [\mathsf{pending@remote}]$$

3 Encoding of Some Distributed Query Patterns

We have seen in Figure 1 how view definitions (either materialized or virtual) and queries are represented in the language. We now show how *distributed query patterns* (in the spirit of design patterns for software engineering) can be described.

Chaining, Recruiting, Referral
These are three facilitating patterns used in software agents architectures and in directory services (LDAP). The three patterns are illustrated in Figure 2-left.

For the sake of illustration, we will assume the existence of three sites: the client, a remote server and the local site where the query process $\mathsf{ch}_1\mathsf{@client} \Leftarrow \mathsf{E}(\mathsf{v@server})$ is pending. The result has to be sent to the client site, while the evaluation of the query requires some data available at the server site.

• *Chaining*

For chaining, the local site will ultimately send the answer to the client. The pending query process is split into two new query processes: the first *qp* expects some data on a local channel ch_2 and forwards it to the client; the second *qp* will evaluate the expression and send it on this local channel. The first *qp* cannot be further installed and is tagged for execution. The second *qp* cannot be further installed locally (it refers to a remote view) and is marked for migration.

$$\xrightarrow{SPLIT} \begin{array}{l} \mathsf{ch}_1@\mathsf{client} \Leftarrow \mathsf{E}(\mathsf{ch}_2@\mathsf{local}) \text{ [pending]} \\ \mathsf{ch}_2@\mathsf{local} \Leftarrow \mathsf{v}@\mathsf{server} \text{ [pending]} \end{array} \begin{array}{l} \xrightarrow{TAG} \mathsf{ch}_1@\mathsf{client} \Leftarrow \mathsf{E}(\mathsf{ch}_2@\mathsf{local}) \text{ [execute]} \\ \xrightarrow{TAG} \mathsf{ch}_2@\mathsf{local} \Leftarrow \mathsf{v}@\mathsf{server} \text{ [pending@server]} \end{array}$$

• *Recruiting*

For recruiting the local site decides to forward the query process to another site (the server in our example). From the client point of view, the answer will come from a site different from the one it addressed the query too. Something to keep in mind is that the local site can rewrite the query before it forwards it.

$$\xrightarrow{TAG} \mathsf{ch}_1@\mathsf{client} \Leftarrow \mathsf{E}(\mathsf{v}@\mathsf{server}) \text{ [pending@server]}$$

Referral is a particular case of recruiting where the recruited site in the originating site.

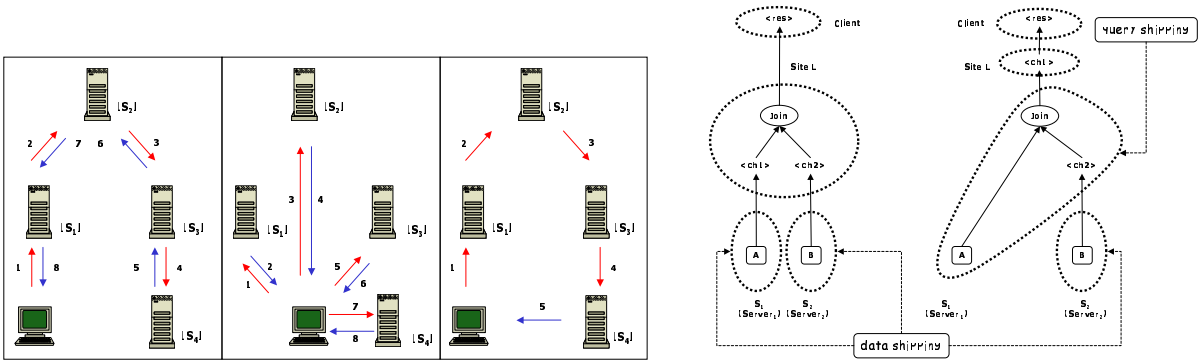

Figure 2: Chaining, Referral, Recruiting (left), Data/Query Shipping (right)

Data/Query/Hybrid Shipping

We now revisit some traditional database techniques for distributed query evaluation, as illustrated in Figure 2-right. The query process $\mathsf{res}@\mathsf{client} \Leftarrow \mathsf{A}@\mathsf{S}_1 \bowtie \mathsf{B}@\mathsf{S}_2$ is pending at the local site (site L). For all cases, the local site will use chaining, i.e. the result will be available locally before being sent to the client.

• *Data shipping*

For data shipping, the local site will perform the join locally and will have the data shipped from sites S_1 and S_2. The local site splits the pending *qp* into three new pending *qp*'s: one to perform the join and one to retrieve each relation from the remote sites. The newly created *qp*'s are then tagged for execution and migration.

$$\xrightarrow{SPLIT} \begin{array}{l} \mathsf{res}@\mathsf{client} \Leftarrow \mathsf{ch}_1@\mathsf{local} \bowtie \mathsf{ch}_2@\mathsf{local} \text{ [pending]} \\ \mathsf{ch}_1@\mathsf{local} \Leftarrow \mathsf{A}@\mathsf{S}_1 \text{ [pending]} \\ \mathsf{ch}_2@\mathsf{local} \Leftarrow \mathsf{B}@\mathsf{S}_2 \text{ [pending]} \end{array} \begin{array}{l} \xrightarrow{TAG} \mathsf{ch}_1@\mathsf{client} \Leftarrow \mathsf{E}(\mathsf{ch}_2@\mathsf{local}) \text{ [execute]} \\ \xrightarrow{TAG} \mathsf{ch}_1@\mathsf{local} \Leftarrow \mathsf{A}@\mathsf{S}_1 \text{ [pending@S}_1\text{]} \\ \xrightarrow{TAG} \mathsf{ch}_2@\mathsf{local} \Leftarrow \mathsf{B}@\mathsf{S}_2 \text{ [pending@S}_2\text{]} \end{array}$$

• *Query shipping*

For query shipping, the local site will decide to have the join evaluated at one of the remote sites. The pending *qp* is split into two new *qp*'s one to get the result and send it to the client and one to evaluate the query remotely. The shipping of the query is done by tagging the *qp* with the join for migration.

$$\xrightarrow{SPLIT} \begin{array}{l} \mathsf{res}@\mathsf{client} \Leftarrow \mathsf{ch}_1@\mathsf{local} \text{ [pending]} \\ \mathsf{ch}_1@\mathsf{local} \Leftarrow \mathsf{A}@\mathsf{S}_1 \bowtie \mathsf{B}@\mathsf{S}_2 \text{ [pending]} \end{array} \begin{array}{l} \xrightarrow{TAG} \mathsf{res}@\mathsf{client} \Leftarrow \mathsf{ch}_1@\mathsf{local} \text{ [execute]} \\ \xrightarrow{TAG} \mathsf{ch}_1@\mathsf{local} \Leftarrow \mathsf{A}@\mathsf{S}_1 \bowtie \mathsf{B}@\mathsf{S}_2 \text{ [pending@S}_1\text{]} \end{array}$$

Hybrid shipping is a combination of both. We can also express other patterns such as subscription, caching, proxying, etc. See the full version of the paper for a description of them.

4 Installation Algorithm and Installation Strategies

In the previous section we have shown how traditional ways to evaluate queries in a distributed setting can be captured using our language. More interestingly, we would like to start with a given configuration (some replicated *qp*'s available at some sites and a query pending at one site) and use our rewrite rules to produce a good execution plan. The installation algorithm can be described as follows:

Installation Algorithm

```
1  while (true)
2   pick the next qp
3     apply MERGE              !! multiple choices !!
4     normalize                (query language dependent)
5     apply SPLIT              !! multiple choices !!
6     apply TAG                !! multiple choices !!
```

What is important to note is that the algorithm is non-deterministic because merge, split and tag usually offer multiple choices. To remove this non-determinism, we need to design some *installation strategies* that resolve these choices. Here are some examples of such installation strategies[1].

Strategy	Description
1	chaining + data-shipping
2	use strategy 1 when # local files > 1, otherwise use recruiting + query-shipping
3	use strategy 1 when #local files > #remote files, otherwise use recruiting + query-shipping
4	hybrid-shipping

In order to evaluate the quality of such installation strategies, we need to be able to compare them to some *"gold-standard"*. In our case, this is the textbook distributed query optimization algorithm [7, 5] based on dynamic programming, adapted for distributed settings as presented in [4]. This algorithm (refer to it as "DP") is guaranteed to find a minimum-cost plan. However, while we wish to compare the plans that our algorithm produces to those obtained by DP, we do not try to use DP itself in ubQL installation. Running DP requires globally complete and timely knowledge of the entire system at each of the query sites where full optimization is attempted. This is not practical for the systems we have in mind. In our approach the installation of a query is spread among several sites, with much fewer alternatives explored. We do not expect to find the minimum cost plan but we hope to find a plan that is not much worse.

To compare against DP, we generate some random configurations and compare the plans produced by DP with the ones produced by our installation strategies. Our random configurations correspond to tree-queries (a generalization of the example of Figure 1) and are defined with multiple layers of mediators, intermediate views and base relations located at various sites. For our experiments, we use 10 base relations, 5 sites and 3 layers of mediators. The query to be installed is a join of the 10 base relations, expressed as a join of some intermediate views. For this experimental configuration, we do not consider redundancy (only one way to merge *qp*'s) and we assume only one query in the system.

For each configuration, we look at 3 scenarios where the size of the relations differ. For the kind of applications we have in mind, it is not realistic to assume accurate cost information. We therefore consider a binary cost-model where relations can be either *small* [2] or *big*. The cost-model we use is defined below.

Cost model		Cardinality
shipping(A)	$50 + \|A\| * 10$	$\|A \bowtie B\| = min(\|A\|, \|B\|)$ if A or B is small
cost($A \bowtie B$)	$3 * max(\|A\|, \|B\|)$	$\|A \bowtie B\| = \|A\| * \|B\| * \frac{1}{3}$ otherwise

The installation strategies mentioned previously (1 to 4) are unfortunately too naive to produce some reasonably good plans: they do not use any cost information. Another strategy (see Figure 3-left) makes use of some binary cost [3] information and produces plans that compare reasonably (see Figure 3-right [4]) to the ones produced by DP.

[1] These strategies resolve *some* choices, but not all.

[2] Small relations are a good way to describe Web queries since a selection can always be represented as the join with a one-row relation.

[3] When using some binary cost information, from a query process point of view, an expression can be described in terms of its local/remote small and local/remote big relations.

[4] These experimental results correspond to the simulated installation of query processes. The simulator is a piece of Java code that applies the various installation strategies.

Note that Strategy 5 produces only one plan (in linear time, compared to exponential time for DP). We conclude that there exists installation strategies using some partial information (binary cost-model and partial local information) that can be successfully used in the ubQL installation algorithm.

Strategy 5

process rewritings
chaining if localBig > 0 and small > 0
chaining if localSmall > 0
recruiting if local $= 0$
recruiting if smallLocal > 0 and bigLocal $= 0$
recruiting if small $= 0$
expression rewirtings
if remote $= 0$ join small relation first
if remoteSmall > 0, hybrid shipping
and join locally with remote small
if remoteSmall $= 0$, hybrid shipping
and ship local small to remote site

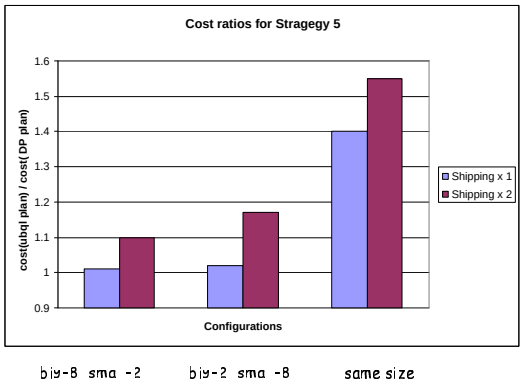

Figure 3: Quality of plans produced using Strategy 5, compared to DP.

5 Conclusion and Future Work

In this paper, we have presented ubQL, a language to describe and deploy distributed queries. The language consists of a set of primitives that can be added on top of any query language. Using the language we can describe well-known distributed query patterns. More interestingly, we have presented an algorithm that permits to rewrite a query and produce a reasonably good evaluation plan, even for local and partial cost information. The quality of such produced plans has been compared with dynamic programming for a mediator-like distributed setting, for single queries and no replication.

In future work, we need to investigate installation strategies that can take into account redundancy in resources and the presence of multiple queries running at the same site. We also intend to enrich the language with process combinators in the spirit of [1] to monitor the execution of a given query and support interleaving execution and installation, yielding a degree of adaptivity [2] to ubQL. Finally, we plan to implement the ubQL primitives on top of XQuery [9].

References

[1] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May/June 1999.

[2] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, and Vijayshankar. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[3] Donald Kossmann. The State of the Art in Distributed Query Processing . Submitted to ACM Computing Surveys.

[4] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM TODS*, March 2000.

[5] Guy M. Lohman, Dean Daniels, Laura M. Haas, Ruth Kistler, and Patricia G. Selinger. Optimization of nested queries in a distributed relational database. In *VLDB*, 1984.

[6] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[7] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 2 edition, 1999.

[8] Tolga Urhan and Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23, June 2000.

[9] W3C. XQuery: A Query Language for XML. W3C Working Draft 15 February 2001. Available from `http://www.w3.org/TR/2001/WD-xquery-20010215/`.