Query Optimization in Distributed Networks of Autonomous Database Systems

FRAGKISKOS PENTARIS and YANNIS IOANNIDIS
Dept. of Informatics and Telecommunications, University of Athens, Hellas

Large-scale distributed environments, where each node is completely autonomous and offers services to its peers through external communication, pose significant challenges to query processing and optimization. Autonomy is the main source of the problem, as it results in lack of knowledge about any particular node with respect to the information it can produce and its characteristics, for example, cost of production or quality of produced results. In this article, inspired by e-commerce technology, we recognize queries as commodities and model query optimization as a trading negotiation process. Subquery answers and subquery operator execution jobs are traded between nodes until deals are struck with some nodes for all of them. Such trading may also occur recursively, in the sense that some nodes may play the role of intermediaries between other nodes (subcontracting). We identify the key parameters of the overall framework and suggest several potential alternatives for each one. In comparison to trading negotiations for e-commerce, query optimization faces unique new challenges that stem primarily from the fact that queries have a complex structure and can be broken into smaller parts. We address these challenges through a particular instantiation of our framework focusing primarily on the optimization algorithms run on "buying" and "selling" nodes, the evaluation metrics of the queries, and the negotiation strategy. Finally, we present the results of several experiments that demonstrate the performance characteristics of our approach compared to those of traditional query optimization.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Distributed databases*; *query processing*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Query optimization

A previous version of this article was published as PENTARIS, F., AND IOANNIDIS, Y. E., Distributed query optimization by query trading, In *Proceedings of the EDBT,* E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, Eds., Lecture Notes in Computer Science, vol. 2992, Springer-Verlag, New York, 2004, 532–550.
This research was partially supported by the Information Society Technologies (IST) Program of the European Commission under the DELOS Network of Excellence on Digital Libraries (Contract G038-507618) and the BRICKS Integrated Project (Contract 507457).
Authors' address: Department of Informatics and Telecommunications, University of Athens, 15785 Ilisia, Athens, Hellas; email:{frank,yannis}@di.uoa.gr.

1. INTRODUCTION

Current requirements on scalability and availability of information foster the formation of large networks of databases with large amounts of data distributed among hundreds or even thousands of autonomous and possibly heterogeneous Database Management Systems (DBMSs). In such environments, finding the answer to a query requires splitting it into parts (subqueries), retrieving the answers of these parts from several remote "black-box" nodes, and merging the results together to calculate the answer of the initial query [Navas and Wynblatt 2001]. Node autonomy and diversity result in lack of knowledge about any particular node with respect to information it can produce and its characteristics, for example, query capabilities, quality of produced results. Earlier work [Deshpande and Hellerstein 2002; Kossmann 2000; Stonebraker et al. 1996] has shown that traditional query optimization techniques [Ioannidis and Kang 1990; Ioannidis et al. 1997; Papadimitriou and Yannakakis 2001] do not work well in federations of autonomous DBMSs. In this article, we consider a new approach to distributed query optimization, particularly suitable for such environments. Inspired by microeconomics, we adapt e-commerce trading negotiation methods to the problem. The result is a query trading mechanism where instead of trading goods, nodes trade answers of (parts of) queries and operator-execution jobs in order to find the best possible distributed query execution plan.

As a motivating example, consider the case of a telecommunications company with hundreds of regional offices. Each of them has a local DBMS, holding customer-care (CC) data of millions of customers. The schema includes the relations customer(*custid*, custname, office), holding customer information such as the regional office responsible for each customer, and invoiceline(*invid*, linenum, custid, amount) holding the details of customers' past invoices. For performance and robustness, each relation may be horizontally partitioned and/or replicated across several regional offices. Assume that a manager at the Athens office asks for the total amount of bills issued at the offices of the Corfu and Myconos islands:

```
SELECT SUM(amount) FROM invoiceline i, customer c
  WHERE i.custid=c.custid AND office in ('Corfu','Myconos');
```

The Athens node will ask all other company nodes whether or not they can evaluate (some part of) the query. Assume that the Myconos and Corfu nodes reply positively about the part of the query dealing with their own customers at a cost of 30 and 40 seconds, respectively. These offers could be based on the nodes actually processing the corresponding queries, or having the results precomputed already, or even receiving them from yet another node; whatever the case, it is of no concern to Athens. It only has to compare these offers against any other it may have and obtain each partial result from the node with the least-cost offer.

This example has Athens effectively *purchasing* the two answers from the Corfu and Myconos nodes at a cost of 30 and 40 seconds, respectively. That is, queries and query-answers are commodities and query optimization becomes a common trading negotiation process. The buyer is Athens and the potential sellers are Corfu and Myconos. The cost of each query-answer is the time to

deliver it. In the general case, the cost may involve several other properties of the query-answers, for example, freshness and accuracy, and may even be monetary. Furthermore, the participating nodes may not be in a cooperative relationship (e.g., members of a company's distributed database) but in a competitive one (e.g., nodes on the Internet offering data products). In the latter case, the goal of each node would be to maximize its private benefits (according to the chosen cost model) instead of the joint benefit of all nodes.

In this article, we present a comprehensive query and processing-task trading negotiation framework, and propose its use as a query optimization mechanism appropriate for large-scale distributed environment of (cooperative or competitive) *purely autonomous* information providers. It is inspired by traditional e-commerce trading negotiation solutions, whose properties have been studied extensively within B2B and B2C systems [Bichler et al. 1999; Collins et al. 1999; Parunak 1987; Sandholm 2002; Su et al. 2001; Winoto et al. 2002], but also by problem solving approaches that distribute tasks over several agents in order to achieve a common goal (e.g., Contract Net [Smith 1980]). Its major differences from these traditional frameworks stem primarily from two facts:

(1) A query is a complex structure that can be broken into smaller parts that can be traded separately. Hence, buyers do not know *a priori* what commodities (query answers) they should buy. Traditionally, only atomic commodities have been traded, for example, a car.
(2) The *value* of a query answer is, in general, multidimensional, for example, system resource consumption, data freshness, data accuracy, response time, etc. Traditionally, only individual monetary values have been associated with commodities.

We focus primarily on the first difference and provide details about the proposed framework with respect to the overall system architecture, negotiation protocols (i.e., bidding, auctions, bargaining), and negotiation contents. We also present the results of several simulation experiments that identify the key parameters affecting query optimization in very large networks of autonomous DBMSs and demonstrate the potential efficiency and effectiveness of our method.

To the best of our knowledge, the only other algorithms suitable for partially autonomous environments are that of the Mariposa system [Stonebraker et al. 1996] and a specific variant of the Iterative Dynamic Programming (IDP) [Deshpande and Hellerstein 2002]. Mariposa has been the first system to ever consider an economic approach to distributed query optimization. Its optimizer is suitable for federations of autonomous databases and works in multiple phases. First, it ignores data distribution to produce locally optimal execution plans; then, it splits these plans to pieces using a heuristic algorithm; finally, it brings back data distribution into consideration and selects execution nodes for the plan pieces using ideas from microeconomics. Our approach requires less information on distant nodes than Mariposa, allowing for higher node autonomy. Furthermore, it performs better by avoiding multiple phases and by

integrating access method selection, query splitting and node selection phases within a unique virtual query and query-answer economy.

In our experimental study, we compare our technique to both the Mariposa query optimizer and IDP and show that it produces better execution plans and, needs (on average) less time to optimize and execute a query. We also experimentally compare our technique to some of the currently dominant centralized techniques for distributed query optimization to show the potential cost of node autonomy.

The remainder of the article is organized as follows: In Section 2, we examine the components of a general trading negotiation framework. In Section 3, we present our query trading technique. In Section 4, we examine the properties of all currently-known algorithms that are relevant to query optimization in distributed networks of autonomous database systems and, in Section 5, we experimentally measure their performance. In Section 6, we present an extended version of our technique supporting both query and processing-task trading and, in Section 7, we discuss the way our framework can support subcontracting. In Section 8, we present some further extensions to our technique. In Section 9, we discuss our findings on distributed query optimization and, in Section 10 we conclude.

The electronic appendix contains information on the structure of network messages exchanged by our technique, implementation details concerning the dynamic programming algorithm used in our experiments, and a discussion on the behavior of our technique when network nodes use advance trading negotiation protocols.

2. TRADING NEGOTIATION FRAMEWORK

A trading negotiation framework provides the means for buyers to request items offered by seller entities. These items can be anything, from plain pencils to advanced gene-related data. The involved parties (buyer and sellers) assign private valuations to each traded item, which in the case of traditional commerce, is usually their cost measured using a currency unit. Entities may have different valuations for the same item (e.g., different costs) or even use different indices as valuations, (e.g., the weight of the item, or a number measuring how important the item is for the buyer). Since valuations are private, sellers will usually make different offers concerning the same item, and therefore, a negotiation procedure will be required before the buyer and sellers reach an acceptable agreement.

Trading negotiation procedures follow rules defined in a negotiation protocol. In each step of the procedure, the protocol designates a number of possible actions (e.g., make a better offer, accept offer, reject offer, etc.). Entities choose their actions, based on the strategy they follow and the expected surplus (utility) from this action, which is defined as the difference between the values agreed in the negotiation procedure and these held privately.

Figure 1 shows the modules required for implementing a distributed electronic trading negotiation framework among a number of network nodes. Each node uses two separate modules, a *negotiation protocol* and a *strategy*

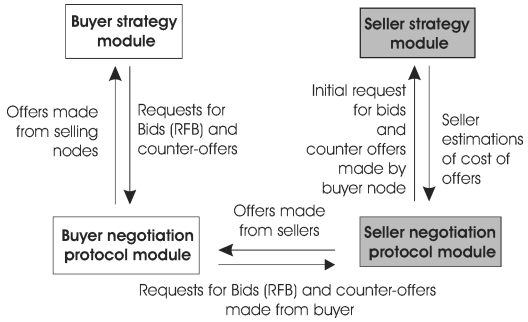


Fig. 1. Modules used in a general trading negotiation framework.

module (white and gray modules designate buyer and seller modules respectively). The former handles the internodes message exchanges and monitors the current status of the negotiation, while the latter selects the contents of each offer/counter-offer. There are many different parameters affecting the behavior of these modules, including which negotiation protocol is used, which mechanism is used to reduce network congestions, what strategy is selected, how the private valuation of the traded items is calculated, and what the contracting details are. We briefly examine the most important of these parameters separately below.

2.1 Negotiation Protocol

The Negotiation protocol designates to the bargaining nodes, the valid actions in each stage of the negotiation (i.e., the rules of the negotiation), including the valid network messages, and how the winner(s) of the negotiations are selected. Negotiation protocols should be efficient, as simple as possible, symmetric (i.e., the same for all nodes), and for scalability reasons, not requiring the use of any centralized decision-making node [Rosenchein and Zlotkin 1994]. In Su et al. [2001], three principal forms of negotiation are defined: *bidding*, *auction*, and *bargaining*.

Bidding is the most simple negotiation protocol. The node interested in purchasing specific items asks for bids from nodes that may be able to supply them. The selling nodes reply with offers that include the properties (e.g. quantity, quality) of the items they can provide. The winning node(s) are the one(s) that make the best offers. The Contract Net Protocol (CNP) [Smith 1980] is a good example of bidding.

Auctions are the second type of negotiation protocol. There are many different auction protocols, which differ in the way prices are quoted and in the manner in which bids are tendered. Among them, the most frequently used is the English (first-price) auction. Auctions are much more complex than direct bidding but have better efficiency in terms of accumulated sellers profit and buyers surplus, and smaller computational cost of the strategy used by sellers [Winoto et al. 2002]. In large markets, using an agents-based auction mechanism substantially reduces the number of network messages exchanged.

Finally, *bargaining* is the third and probably the oldest negotiation mechanism. The buyer and sellers make alternating offer/counter-offers until an agreement is reached. The bargaining protocol is not as simple as the bidding protocol, and the strategy used is more complex than that of auctions, yet it is the only one allowing the negotiation of every property of an offer using counter-offers, like for instance, how may items will be purchased, their quality, and the delivery date. Bargaining with a large number of nodes is considered inefficient.

2.2 Strategies

The negotiation protocol designates the actions that are valid during the negotiation process. For instance, in an English auction, any participating node may bid a higher price than the currently winning offer, do nothing or leave the auction. The strategy each node follows is *the set of rules that designates the exact action the node will choose depending on the knowledge it has about the other nodes.* For instance, a buyer may *strategically* delay to accept an offer hopping that in the mean while the seller will improve its offer.

Both buyers and sellers have their private valuation of the bargained items. The buyers/sellers will not accept any offer or make any counter-offer above/below their private valuation. A proper strategy should ensure that seller's and buyer's offers converge towards a common acceptable value.

Traditionally, strategies are classified as either *cooperative* or *competitive* (*noncooperative*). *In the former, the involved entities aim to maximize the joint surplus of all parties, whereas in the latter, they try to individually maximize their personal utility*.

2.3 Offers Valuation

In order for a strategy to work, there must be some way to rank two offers. The designer of the trading mechanism can define certain functions that map the values of the qualitative properties of the offers into real numbers, in such a way that better values are mapped into larger (or, depending on the parameter, smaller) numbers, and then specify a weighting *aggregation* function that maps these numbers into a single one measuring how good an offer is.

3. DISTRIBUTED QUERY TRADING FRAMEWORK

Using the e-commerce negotiation paradigm, we have constructed an efficient algorithm for optimizing queries in large disparate and autonomous environments. Although our framework is more general, in this paper, we limit ourselves to select-project-join queries. The optimization algorithm works by gradually allocating queries and their processing-tasks to remote nodes. In this section, we explain the framework for trading queries focusing on the parts of the general trading framework we have modified, and leave the explanation of processing-tasks outsourcing to Section 6. The reader can find additional information on parts that are not affected by our algorithm, such as general competitive strategies and equilibriums, message congestion protocols, and details on negotiation protocol implementations in Bichler et al. [1999], Collins et al. [1999], Conitzer and Sandholm [2003], Ogston and Vassiliadis [2002],

Parunak [1987], Sandholm [2002], Su et al. [2001], and Winoto et al. [2002] and on standard e-commerce and strategic negotiations textbooks (e.g., Kagel [1995], Kraus [2001], and Rosenchein and Zlotkin [1994]). Furthermore, there are possibilities for additional enhancements of the algorithm that will be covered in future work. These enhancements include the use of *contracting* to model partial/adaptive query optimization techniques and the examination of various *competitive* strategies that lead to Pareto optimal load balancing schemas.

3.1 Query Trading Overview

The idea of the basic query trading (QT) algorithm is to consider queries and query-answers as commodities and the query optimization procedure as a trading of query-answers between nodes holding information that is relevant to the contents of these queries. Buying nodes are those that are unable to answer some query, either because they lack the necessary resources (e.g., data, I/O, CPU), or simply because outsourcing the query is better than having it executed locally. Selling nodes are the ones offering to provide data relevant to some parts of these queries. Each node may play either role (buyer and seller) depending on the query been optimized and the data that each node locally holds.

Before proceeding with the presentation of the optimization algorithm, we should note that no query or part of it is physically executed during the whole optimization procedure. The buyer simply asks from sellers for assistance in evaluating some *queries* and sellers make offers that contain their *estimated* properties of the answer of these queries (*query-answers*). These properties can be the total time required to execute and transmit the results of the query back to the buyer, the time required to find the first row of the answer, the average rate of retrieved rows per second, the total rows of the answer, the freshness of the data, the completeness of the data, and possibly a charged amount for this answer. The query-answer properties are calculated by the sellers' query optimizer and strategy modules, therefore, they can be extremely precise, taking into account the available *network resources* and the *current workload* of sellers.

The buyer ranks the offers received using an administrator-defined weighting aggregation function (see Section 2.3) and chooses those that minimize the total cost/value of the query. In the remaining of this section, the valuation of the offered query-answers will be the total execution time (cost) of the query, thus, we will use the terms cost and valuation interchangeably. However, nothing forbids the use of a different cost unit, such as the total network resources used (number of transmitted bytes) or even monetary units.

3.2 The Query-Trading Algorithm

The execution plans produced by the query-trading (QT) algorithm consist of the query-answers offered by remote sellers together with the processing operations required to construct the results of the optimized queries from these offers. The task of the algorithm is to find the combination of data offers and buyer and seller processing operations that minimize the valuation (cost) of the final answer. For this reason, it runs iteratively, progressively selecting the best execution plan. In each iteration, the buyer asks (Request for Bids—RFBs) for

Buyer-side algorithm	**Sellers-side algorithm**
B0. Initialization, set $Q = \{\{q, C\}\}$ B1. Make estimations of the values of the queries in set Q, using a trading strategy. B2. Request offers for the queries in set Q	S1. For each query q in set Q do the following: S2.1. Find sub-queries q_k of q that can be answered locally. S2.2. Estimate the cost c_k of each of these sub-queries q_k. S2.3. Find other (sub-)queries that may be of some help to the buyer.
B3. Select the best offers $\{q_i, c_i\}$ using **one of the three** methods (bidding, auction, bargaining) of the query trading framework	S3. Using the query trading framework, make offers and try to sell some of the subqueries of step S2.2 and S2.3.
B4. Using the best offers, find possible execution plans P_m and their estimated cost C_m B5. Find possible sub-queries q_e and their estimated cost c_e that, if available, could be used in step B4. B6. Update set Q with sub-queries $\{q_e, c_e\}$. B7. Let P_* be the best of the execution plans P_m. If P_* is better than that of the previous iteration of the algorithm, or if step B6 modified the set Q, then go to step B1. B8. Inform selling-nodes, which queries are used in the best execution plan P_*, so that they start executing these queries.	

Fig. 2. The query trading (QT) algorithm.

some queries and the sellers reply with offers that contain the estimations of the properties of these queries (query-answers). Since sellers may not have all the data referenced in a query, they are allowed to give offers for only the part of the data they actually have. At the end of each iteration, the buyer uses the received offers to find the best possible execution plan, and then, the algorithm starts again with a possibly new set of queries that might be used to construct an even better execution plan.

The optimization algorithm is actually a kind of bargaining between the buyer and the sellers. The buyer asks for certain queries and sellers counter-offer to evaluate some (modified parts) of these queries at different values. The difference between our approach and the general trading framework, is that in each iteration of this bargaining the negotiated queries are different, as the buyer **and** sellers progressively identify additional queries that may help in the optimization procedure. This difference, in turn, makes necessary to change selling nodes in each step of the bargaining, as these additional queries may be better offered by other nodes. This is in contrast to the traditional trading framework, where the participants in a bargaining remain constant.

Figure 2 presents the details of the distributed optimization algorithm. The input of the algorithm is a query q with an initially estimated cost of C. If no estimation using the available local information is possible, then C is a

predefined constant (zero or something else depending on the type of cost used). The output is the estimated best execution plan P_* and its respective cost C_* (step B8). The algorithm, at the buyer-side, runs iteratively (steps B1 to B7). Each iteration starts with a set Q of pairs of queries and their estimated costs, which the buyer would like to purchase from remote nodes. In the first step (B1), the buyer strategically estimates the values it should ask for the queries in set Q, and then asks for bids (RFB) from remote nodes (step B2). The (candidate) sellers after receiving this RFB make their offers, which contain query-answers concerning parts of the queries in set Q (step S2.1–S2.2) or other relevant queries that could be of some use to the buyer (step S2.3). The winning offers are then selected using a small nested trading negotiation procedure (steps B3 and S3). The buyer uses the contents of the winning offers to find a set of candidate execution plans P_m and their respective estimated costs C_m (step B4), and an enhanced set Q of queries-costs pairs (q_e, c_e) (steps B5 and B6) which they could possibly be used in the next iteration of the algorithm for further improving the plans produced at step B4. Finally, in step B7, the best execution plan P_* out of the candidate plans P_m is selected. If P_* is not better than that produced in the previous iteration (i.e., no further improvement is possible) and step B5 did not find any new query, then the algorithm is terminated.

As previously mentioned, our algorithm looks like a general bargaining with the difference that in each step the sellers and the queries bargained are different. In steps B2, B3 and S3 of each iteration of the algorithm, a complete (nested) trading negotiation is conducted to select the best sellers and offers. The protocol used can be any of the ones discussed in Section 2.1. If the expected number of offers is large, then an auction mechanism should be preferred, otherwise, a bidding or bargaining protocol should be used. More specifically, if only minor modifications in the offers are required (e.g., a change of a single query-answer property), then the use of bargaining will improve the performance as it will help avoid a costly algorithm iteration. If this is not the case, like for instance when a modification of a query structure is required, then bidding should be preferred. This is because after such a modification, the algorithm will run at least one more iteration and since each iteration is actually a generalized bargaining step, using a nested bargaining within a bargaining would only increase the number of exchanged messages.

3.3 Algorithm Details

Figure 3 shows the modules required for an implementation of our query trading algorithm (grayed boxes are modules running at the sellers) and the processing workflow between them. As Figure 3 shows, the buyer initially assumes that the value of query q is C, and asks its *buyer strategy* module to make a (strategic) estimation of its value using a traditional e-commerce trading reasoning. This estimation is given to the *buyer negotiation protocol* module that asks for bids (RFB) from distant sellers. The latter use their *seller negotiation protocol* module to receive this RFB and forward it to their *partial query constructor and cost estimator* module, which builds pairs of a possible part of query q together with an estimate of its respective value. The pairs are forwarded to the *seller predicates analyzer* to examine them and find additional queries (e.g.,

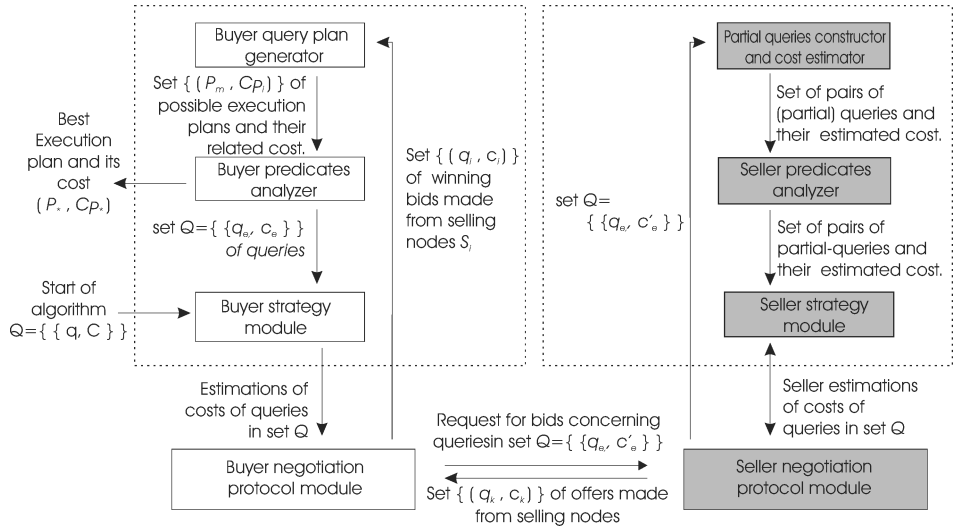


Fig. 3. Modules used by the query trading algorithm.

materialized views) that might be useful to the buyer. The output of this module (set of (sub-)queries and their costs) is given to the *seller strategy module* to decide (using again an e-commerce trading reasoning) which of these pairs is worth attempting to sell to the buyer, and in what value. The *negotiation protocol* modules of both the sellers and the buyer then run though the network a predefined trading protocol (e.g., bidding) to find the winning offers (q_i, c_i). These offers are used by the buyer as input to the *buyer query plan generator*, which produces a number of candidate execution plans P_m and their respective buyer-estimated costs C_m. These plans are forwarded to the *buyer predicates analyzer* to find a new set Q of queries q_e and then, the workflow is restarted unless the set Q was not modified by the *buyer predicates analyzer* and the *buyer query plan generator* failed to find a better candidate plan than that of the previous workflow iteration.

It is worth comparing Figure 1, which shows the typical trading framework, to Figure 3, which describes our query trading framework. These figures show that the buyer strategy module of the general framework is enhanced in the query trading framework with a query plan generator and buyer predicates analyzer. Similarly, the seller strategy module is enhanced with a partial query constructor and a seller predicates analyzer. These additional modules are required, since in each bargaining step the buyer and the sellers make (counter-) offers concerning a different set of queries, than that of the previous step.

To complete the analysis of the distributed optimization algorithm, we examine in detail each of the modules of Figure 3, starting from the seller ones.

3.4 Partial Query Constructor and Cost Estimator

The role of the partial query constructor and cost estimator of the selling nodes is to construct a set of queries q_k offered to the buyer. It uses two separate algorithms, explained below:

Sellers may not have all necessary base relations, or relations' partitions, to process all queries asked by the buyer (set Q). Therefore, they have to examine each query q_e of Q and rewrite it (if possible), using the following algorithm, which removes all non-local relations and restricts the base-relation extents to the locally-available partitions. After trimming the queries, they are split (if necessary) into pieces in such a way that no Cartesian products exist. The split queries are stored in a set CQ, which is the output of the algorithm.

Query rewrite algorithm

1 While there is a relation R in query q_e that is not available locally, do
2 Remove R from q_e.
3 Update the SELECT-part of q_e adding the attributes of the remaining relations of q_e that are used in joins with R.
4 Update the WHERE-part of q_e removing any operators referencing relation R.
5 End While
6 For each remaining relation R in query q_e, update the WHERE-part of q_e adding the restriction operators of partitions of R stored locally.
7 Let set $CQ = \emptyset$. Insert q_e into CQ.
8 While set CQ contains a query with Cartesian product, do
9 Let q' be the first query of CQ with Cartesian product.
10 Remove q' from CQ.
11 Break q' into two pieces q'_1 and q'_2 so that q'_1 contains no Cartesian product. Query q'_1 will contain all relations of q' that have a mutual join condition in the WHERE-part of q'. If no such relations exist, q'_1 will contain the first relation of q'. q'_2 will contain the remaining relations of q'.
12 Insert q'_1 and q'_2 into CQ.
13 End While

As an example of how the previous algorithm works, consider the example of the telecommunications company and consider again the example of the query asked by the manager at Athens. Assume that the Myconos node has the whole `invoiceline` table but only the partition of the customer table with the restriction `office='Myconos'`. Then, after running the query rewriting algorithm at the Myconos node and simplifying the expression in the WHERE part, the resulting query in set CQ will be the following:

```
SELECT SUM(amount) FROM invoiceline i, customer c
WHERE i.custid=c.custid AND office='Myconos';
```

The restriction `office='Myconos'` was added to the above query, since the Myconos node has only this partition of the customer table.

After running the query rewrite algorithm, sellers use their local query optimizer to find the best possible local plan for each (rewritten) query of set CQ. This is needed to estimate the properties and cost of the query-offers they will make. Conventional local optimizers work progressively pruning suboptimal access paths, first considering two-way joins, then three-way joins, and so on, until all joins have been considered [Selinger et al. 1979]. Since, these partial results may be useful to the buyer, we include the optimal two-way, three-way, etc. partial results in the offer sent to the buyer. The modified dynamic

programming (DP) algorithm [Halevy 2001] that runs for each (rewritten) query q is the following (The queries in set D are the result of the algorithm):

	Modified DP algorithm
1	Find all possible access paths of the relations of q.
2	Compare their cost and keep the least expensive.
3	Add the resulting plans into set D.
4	For i = 1 to number of joins in q, do
5	Consider joining the relevant access paths found in previous iterations using all possible join methods.
6	Compare the cost of the resulting plans and keep the least expensive.
7	Add the resulting plans into set D.
8	End For

If we run the modified DP algorithm on the output of the previous example, we will get the following queries:

```
1. SELECT custid FROM customer WHERE office='Myconos';
2. SELECT custid,amount FROM invoiceline;
3. SELECT SUM(amount) FROM invoiceline i, customer c
   WHERE i.custid=c.custid AND office='Myconos';
```

The first two SQL queries are produced in steps 1–3 of the dynamic programming algorithm and the last query is produced in the first execution (i = 1) of steps 5–7.

3.5 Seller Predicates Analyzer

The seller predicates analyzer works complementarily to the partial query constructor to further find queries that might be of some interest to the buyer. The latter is based on a traditional dynamic programming optimizer and therefore does not necessarily find all queries that might be of some help to the buyer. If there is a materialized view that might be used to quickly find a superset/subset of a query asked by the buyer, then it is worth offering (in small value) the contents of this materialized view to the buyer. For instance, continuing the example of the previous section, if Myconos node had the materialized view:

```
CREATE VIEW invoice AS
   SELECT custid, SUM(amount) FROM invoiceline GROUP by custid;
```

then it would worth offering it to the buyer, as the grouping asked by the manager at Athens is more coarse than that of this materialized view. There are a lot of non-distributed algorithms concerning answering queries using materialized views with or without the presence of grouping, aggregation and multidimensional functions, like for instance [Zaharioudakis et al. 2000]. All these algorithms can be used in the seller predicates analyzer to further enhance the efficiency of the QT algorithm and enable it to consider using remote materialized views. The potential of improving the distributed execution plan by using materialized views is substantial, especially in large databases, data warehouses and OLAP applications.

3.6 Buyer Query Plan Generator

The query plan generator combines the queries q_i that won the bidding procedure to build possible execution plans P_m for the original query q. The problem of finding these plans is identical to the answering queries using materialized views [Pottinger and Levy 2000] problem. In general, this problem is NP-Complete, since it involves searching though a possibly exponential number of rewritings.

The most simple algorithm that can be used is the dynamic programming (DP) algorithm that we describe below. Its input are the queries q_i and the output of the algorithm is the set D of candidate plans P_m. If query q is the one been optimized, then candidate plans are those representing an equivalent rewriting of query q using some of the queries q_i that won the bidding procedure (see Figure 3):

	DP buyer plan generator
1	Find all queries q_i that are candidate execution plans and move them to set D.
2	Compare the cost of the remaining (partial) plans and keep those for which there is no other partial plan with the same or greater contribution towards query q and equal or smaller value.
3	While there are still some partial plans left, do
4	Consider joining or unioning all remaining partial plans using all possible ways.
5	Find all candidate execution plans and move them to set D.
6	Compare the cost of the remaining (partial) plans and keep those for which there is no other partial plan with the same or greater contribution towards query q and equal or smaller value. The value of two plans can be compared only if they run on the same set of nodes and their results are delivered on the same node.
7	End While

In the experiments presented in the next sections, apart from the DP algorithm, we have also considered the use of a version of the Iterative Dynamic Programming IDP-M(3,5) algorithm proposed in Kossmann and Stocker [2000]. This algorithm is similar to DP. Its only difference is that after evaluating all 3-way join subplans, it keeps the best five of them throwing away all other 3-way join subplans, and then it continues processing like the DP algorithm.

If the buyer plan generator cannot aggregate a plan from the offers received then:

(1) If the environment is competitive this may be caused by some malicious sellers trying to increase the value of their offers. It is the responsibility of the buyer strategy module to decide whether the buyer should re-issue the RFB or abort. Note that the strategy module will not always re-issue the RFB, as this will increase the tendency of sellers to ignore the first RFB (and thus save resources and possibly increase their profit), especially if they know they are the only one capable of supplying certain data.

(2) If the environment is cooperative then this may be caused by a temporary outage of resources from a seller that is the only one supplying the data requested. For this reason, the algorithm runs one more iteration before aborting.

3.7 Buyer Predicates Analyzer

The buyer predicates analyzer enriches the set Q (see Figure 3) with additional queries, which are computed by examining each candidate execution plan P_m (see previous subsection). If the queries used in these plans provide redundant information, it updates the set Q adding the restrictions of these queries which eliminate the redundancy. Other queries that may be added to set Q are simple modifications of the existing ones with the addition/removal of sorting predicates, or the removal of some attributes that are not used in the final plan.

To make the functionality of the buyer predicate analyzer more concrete to the reader, consider again the telecommunications company example, and assume that someone asks the following query:

```
SELECT custid FROM customer WHERE office in ('Corfu','Myconos','Santorini')
```

Assume that one of the candidate plans produced from the buyer plan generator contains the union (distinct) of the following queries:

```
1. SELECT custid FROM customer WHERE office in ('Corfu', 'Myconos');
2. SELECT custid FROM customer c WHERE office in ('Santorini', 'Myconos');
```

The buyer predicates analyzer will see that this union has redundancy and will output the following three queries:

```
1. SELECT custid FROM customer WHERE office='Corfu';
2. SELECT custid FROM customer WHERE office='Myconos';
3. SELECT custid FROM customer WHERE office='Santorini';
```

In the next iteration of the algorithm, the buyer will also ask for bids concerning the above three SQL statements, which will be used in the next invocation of the buyer plan generator, to build a redundancy-free plan that is equivalent to the above union-based one.

3.8 Protocol and Strategy Modules

The protocol and strategy modules are responsible for implementing the trading negotiation protocol (see Section 2.1) and handling the message exchanges between buyers and sellers. All of the previously mentioned negotiation protocols can be used. In most cases, the bidding protocol will provide the best performance. If an agent-based architecture can be used, then an even better alternative is to use an auction mechanism. Finally, bargaining can be used in certain cases to reduce then number of iterations of the QT algorithm.

In this article, we consider a simple cooperative seller strategy, where sellers, if possible, make bids that contain the true valuation of the offered queries. However, all of our findings are applicable in case where QT is used in a competitive environment as well. Designing competitive strategies is beyond the scope of this article, as it requires extensive knowledge of game theory.

4. DISTRIBUTED QUERY OPTIMIZATION ALGORITHMS

There are serveral algorithms that have been proposed in the past for distributed query optimization. In this section, we outline the most prominent

such algorithms and discuss several key details. The following section describes the results of their experimental evaluation.

4.1 Node Autonomy and Amount of Published Information

Each of the distributed query optimization algorithms examined has different requirements with respect to the characteristics of each node that must be publicly available for the algorithm's operation. This essentially defines the level of node autonomy that can be afforded by each algorithm. The types of information that may or may not be required are described below:

Capabilities. Each node may allocate different amounts of processing, main memory and I/O resources for use by distant nodes. In addition, it may have different query processing capabilities. For instance, in our experiments, we have assumed that all nodes supported nested-loops and sort-merge joins but only 80% of them also supported hash-joins.

Information on node capabilities is required by some optimization algorithms to calculate the cost of operations and ensure that certain parts of query execution plans can be evaluated on specific nodes. Publishing this information harms node autonomy since administrators must inform about any modifications that alter node capabilities (e.g., software updates, installation of more memory, etc.).

Logical Schema. The tables stored locally at each node is the minimum information required for any query optimization algorithm to work. It is used to find candidate sources of data required by a query. Making it public requires announcement of all `CREATE/DROP TABLE` DDL statements executed by each node.

Physical Schema—Partitioning. For performance reasons, it is common for administrators of large databases to horizontally partition relations and distribute their parts across the network. Subsequently, they constantly modify the resulting partitioning schemas to ensure optimal performance.

Some optimization algorithms require partitioning information to optimally split join operations across multiple machines. Making this information public requires announcement of all `CREATE/DROP/MERGE/MODIFY PARTITION` DDL statements executed.

Physical Schema—Indices. The set of relation indices that exist in a node may be used by some optimization algorithms to improve the quality of query execution plans produced. Making this information public requires announcement of all `CREATE/DROP INDEX` DDL statements executed.

Data Statistics. Histograms or other approximations of the data distributions in a node may be used by some optimization algorithms to estimate operator selectivity. Making this information public requires announcement of all DML statements that substantially modify data statistics.

Workload. This information is used by some optimization algorithms to estimate the cost of operations by understanding how system resources will be split. The workload is affected by all DML statements, for example,

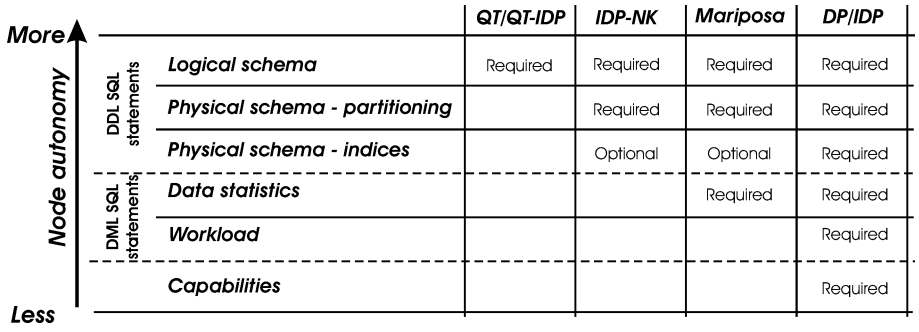


Fig. 4. Types of information required by the algorithms tested.

`SELECT/INSERT/UPDATE/DELETE` (and some DDL statements, for example, `CREATE/DROP/ALTER TABLE/INDEX/PARTITION`). Making this information public requires announcement of almost every activity of distant nodes, which is not practical. Alternatively, it may be estimated using techniques from the area of mediation systems.

The less information an algorithm needs, the more suitable for autonomous environments it is. In the next section, we describe the algorithms tested and compare their needs for knowledge.

4.2 Query Optimization Algorithms

In this article, we study the following six algorithms: Dynamic Programming (DP), Iterative Dynamic Programming (IDP), IDP with minimum knowledge (IDP-NK), Mariposa, and two variations of Query Trading (QT and QT-IDP). Figure 4 summarizes the needs of these six algorithms for public knowledge. The algorithms are further elaborated below:

Dynamic Programming (DP). The Dynamic Programming (DP) algorithm [Selinger et al. 1979] progressively builds plans with 1,2, ..., $n-1$ joins of tables, where n is the number of tables in the query being optimized. DP is intended to run in *centralized* environments, as it requires complete information to run (see Figure 4) and, hence, is unable to respect node autonomy. In the experiments that we have run, we saw that in some cases there was not enough system memory to complete optimization of certain queries. This has been solved by allowing our implementation of DP to automatically fall back to the IDP algorithm (described below) when memory usage exceeded a specific threshold. This is explained in detail in Section B.3 of the electronic appendix.

DP is examined because it produces optimal execution plans and, hence, serves as a solid basis for comparing the quality of plans produced by the remaining algorithms. Our implementation assumed nodes running the algorithm had exact knowledge of the state of the whole network. This avoided all network traffic, which would normally dominate its execution time, but rendered the resulting optimization times for DP not directly comparable to those algorithms respecting some levels of node autonomy.

Iterative Dynamic Programming (IDP and IDP-NK). The Iterative Dynamic Programming (IDP) family of algorithms [Kossmann and Stocker 2000;

Deshpande and Hellerstein 2002] is a heuristic extension of DP that exhibits better performance at the expense of slightly increased cost for the resulting plan. Given an n-way join query, IDP(k,m) first enumerates all subqueries that contain less than or equal to k base tables and finds their costs, following the DP algorithm. Then, it chooses the best m sub-plans out of those produced for all k-way joins together, and purges all others. Finally, it resumes the DP optimization procedure for larger subqueries by examining only the subplans chosen.

For our study, we have implemented two variations of IDP(3,5). The first one (IDP) is the original *centralized* form and the direct counterpart of DP above. It assumes that nodes running the algorithm have exact knowledge of the state of the whole network and, thus, avoids the bottleneck of network congestion. It is studied because it produces plans that are close to the optimal plans of DP [Kossmann and Stocker 2000] and yet it requires substantial less memory than DP, making it particularly well suited for distributed environments [Deshpande and Hellerstein 2002].

The second variation of IDP (IDP-NK) follows an earlier suggestion [Deshpande and Hellerstein 2002] and decouples cost estimation from the query optimization process itself by delaying pruning of plans until the end of each step of IDP. In each step, IDP-NK contacts data providers using a single round of communication to find the exact cost of each execution plan and then uses this information to prune suboptimal plans. Figure 4 shows that IDP-NK requires only relation location and partitioning information to work, making it more suitable for autonomous environments than IDP. Knowledge of relation indices is optional for the operation of IDP-NK, but enables the algorithm to produce plans equivalent in quality to those produced by plain IDP. In our tests, such information was indeed available to IDP-NK.

Mariposa. The Mariposa Distributed DBMS [Stonebraker et al. 1994] addressed the question of distributed query optimization in autonomous systems, and was the first one to propose a microeconomics-based solution for this problem. Its optimizer is a two-phase algorirhm that considers conventional optimization factors (such as join orders) separately from distributed system factors (such as data layout and execution location). First, it uses locally-kept information about various aspects of the data and constructs a locally optimal plan by running a DP optimizer over the query, disregarding the physical distribution of the base relations and fixing such items as join order and data access methods. It then uses network yellow-pages (directory service) information to parallelize query operators, and a bidding protocol together with a greedy algorithm to select operators' execution sites, all in a single interaction with remote nodes. The degree of parallelism is statically determined by the system administrator before query execution and is independent of the distributed resources available.

The Mariposa System was initially designed to cover a large range of different requirements. In this paper, we have implemented the particular description of the Mariposa algorithm presented in the Mariposa database management system [2002]. In the second greedy phase of the algorithm, the optimization

goal was set to minimize the total execution time of the query, which is the task of interest in our experiments. We have not used Mariposa's MERGE operator (implementing parallel k-way data merges and joins), whose presence is not relevant to our analysis.[1] Finally, the Mariposa algorithm was configured to use the maximum possible degree of parallelism, that is, joins were split to the maximum possible number of pieces. This setting is a commonly accepted solution for distributed optimization [Liu and Rundensteiner 2005] and gave the best overall measurements for the Mariposa algorithm.

We have included Mariposa in our study as it is one of the fastest-running known algorithm for distributed query optimization. It is suitable for autonomous environments, although it requires more information for its first phase than QT and IDP-NK (Figure 4). It needs data statistics and, optionally, information on indices [Deshpande and Hellerstein 2002] to produce execution plans of higher quality. In its second phase, it needs partitioning information to properly split and parallelize join operations. Note that indices are used if they are the same for all partitions of a relation, as in the first phase partitioning information is ignored. In our experiments, when relations had multiple partitions, we respected node autonomy and let them build potentially different indices per partition.

Query Trading. We have instantiated the Query Trading Algorithm using the following properties:

For the Negotiation protocol, we chose the bidding protocol (see Section 2.1) as the expected number of offered bids for each RFB was not large enough for an auction protocol to be useful. We ranked each offer based only on the time required to return the complete query answer. In this way, our optimization algorithm produced plans that had the minimum execution time, reflecting traditional optimization.

We used a plain cooperative strategy. Thus nodes replied to RFBs with offers matching exactly their available resources. Note that in Pentaris and Ioannidis [2004], the seller strategy module had a small buffer that held the last 1,000 accepted bids. These bids were collected from past biddings that the node had participated in. Using the contents of this buffer, nodes estimated the most probable value of the offer that will win a bidding and never made offers with value more than 20% of this estimation. This strategy helped to reduce the number of exchanged messages but made QT produce worse plans as it caused buyers to lose some useful plans. Thus in this paper, we chose to disable this mechanism.

In the *buyer query plan generator* we used a traditional answering-queries-using-views dynamic programming algorithm. However, to decrease its complexity we also tested the plan generator with the IDP-M(3,5) algorithm (QT-IDP).

[1]In Pentaris and Ioannidis [2004], Mariposa did use the MERGE operator, but for the sake of equality to the remaining algorithms, we have decided to disable it in the experiments presented in this article.

Table I. Simulation Parameters

Parameter Type	Parameter	Value
Network	Total size of network (WAN)	5,000 nodes
	WAN network packet latency	1–30 ms (uniform distribution)
	WAN interconnection speed	10–100 Mbits/s (uniform distribution)
	Directory service	centralized/DHT-based
RDBMS	Join capabilities	Nested-loops, merge-scan and Hash-join
	Operators pipelining support	Yes
	CPU resources	One simulated 2–4 GHz CPU (uniform distribution)
	Sorting/Hashing buffer size	10,000 tuples
	I/O speed per node	5–40 Mbytes/s (uniform distribution)
	Soft memory usage limit during optimization	10K execution plans (approximately 10–100 MBytes)
	Maximum memory used during optimization	Up to OS limit (2 GByte)
Dataset	Number of relations	10,000
	Size of relations	1-400,000 tuples (uniform distribution)
	Number of attributes per relation	10 attributes
	Number of partitions per relation	1–9
	Number of mirrors per partition	0–29
	Indices per partition per node	2 single-attribute indices
Workload	Joins per query	0–29
	Partitions per relation	1–9
	Shape of queries	path/star

Since there was no limit on the time the buyer waited for offers and the seller strategy did not block offers with low probability of being accepted (see, for instance, Pentaris and Ioannidis [2004]), no offers were lost. Thus, if relations were not partitioned or mirrored, then the QT and plain DP examined the same plans and created the same execution plans. Otherwise, QT evaluated fewer plans than plain DP, as the latter examined all possible data sources and all ways of horizontally splitting the joins. QT examined only those that were possible based on the offers it received and accepted from the sellers.

5. EXPERIMENTAL STUDY

In order to assert the quality of QT, we have simulated a large network of interconnected RDBMSs and run several experiments to measure the performance of our proposed solution in comparison to the other distributed query optimization algorithms mentioned earlier. The details of the study are elaborated in the rest of this section.

5.1 Experiment Setup

5.1.1 *Experiment Parameters.* We have used C++ to build a simulation of a large Wide Area Network (WAN). The parameters of this environment, together with their possible values, are displayed in Table I. All range parameters followed the uniform distribution. The network had 5,000 nodes that exchanged messages with a simulated latency of 1–30 ms and a speed of 10–100 Mbits/s.

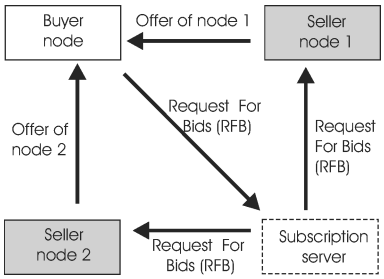


Fig. 5. Centralized directory service architecture.

Each node was equipped with a single CPU at 2-4 GHz that hosted an RDBMS capable of evaluating joins using the nested-loops and the merge-scan algorithms. 80% of the simulated RDBMSs could also use the hash-join method. The local I/O speed of each node was not constant and varied between 5 and 40 MBytes/s. The only (hard) limit on the amount of memory consumed by the optimizer was the one forced by the operating system used (2 GBytes).

The data-set used in the experiment was synthetically built. We constructed the metadata of a large schema consisting of 10,000 relations. There was no need to actually build the data since our experiments measured the performance of query optimization, not that of query execution. Query execution times were estimated from the statistics of the relations. Each relation had 1-400,000 tuples and was horizontally range-partitioned to 1-9 disjoint partitions stored in possibly different nodes in such a way that each partition had up to 29 mirrors. The nodes were allowed to create 2 local indices per locally-stored partition.

5.1.2 *Directory Service Architecture.* Execution of QT, QT-IDP, Mariposa, and IDP-NK algorithms assumes the existence of a directory service (yellow pages) holding the corresponding information that each algorithm requires, as specified in Figure 4 (including all information indicated as optional). This directory service can be implemented in either a centralized or a distributed (P2P) fashion. The latter is most likely used in cases where RFBs' arrival frequency is too much to be handled by a single centralized system. For completeness reasons, we have evaluated both solutions.

As a centralized solution, we have implemented a publish-subscribe mechanism, since these are widely used in existing agent-based e-commerce platforms [Ogston and Vassiliadis 2002] and have good scalability characteristics. All nodes registered information concerning their locally-stored relations to a special subscription server. Figure 5 describes how this server was used by QT during query optimization. The lines show the flow of network message exchanges. Buyers sent to the subscription server request for bids (RFB) messages containing, among others, the description of the queries requested. The server examined the FROM-part of these queries and relayed these RFBs to all sellers having relevant data. Note that the subscription server did not modified the query descriptions relayed. This was done by the sellers' *partial query constructor and cost estimator* explained in Section 3.4. Finally, sellers communicated their replies to RFBs (offers) directly to the buyers (i.e., the did not sent

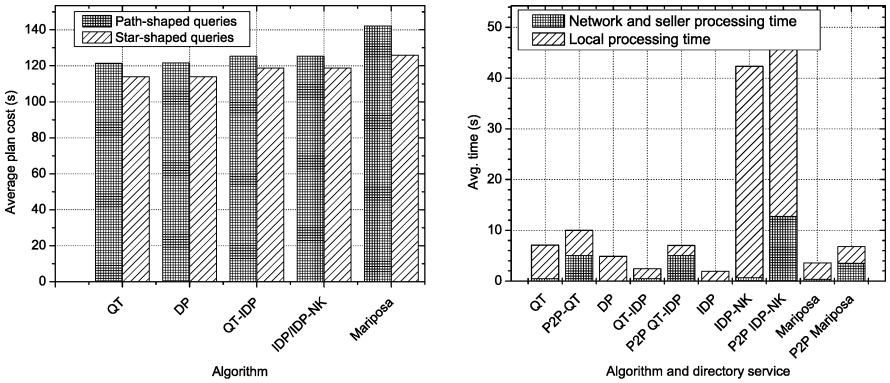
the replies though the subscription server). The overall cost of each bidding in terms of network resources was two messages per seller plus one message from the buyer to the subscription server. There was no limit on the time required for a bidding procedure to complete. The remaining algorithms operated in a similar fashion according to their needs.

To test the algorithms using a distributed directory, we simply replaced the centralized subscription server of Figure 5 with a simulated DHT-based directory. The DHT was based on the Chord protocol [Stoica et al. 2001]. The lookup of keys required $log(N)$ messages, where N was the number of network nodes. Each node for each locally-held relation inserted into the DHT a pair of <relationName, relationInformationStruct>, that is, the name of the relation was the key of the DHT and data were relationInformationStruct structures holding information on relation's locations, partitions, indices and statistics. All RFBs were routed to the relevant seller nodes through the DHT routing tables. The keys searched were the relations referenced in the queries' FROM-part contained in RFBs.

5.2 Simulated Scenarios

We initially ran some experiments to assert the scalability of our algorithm in terms of network size. As expected, the performance of our algorithm was not dependent on the total number of network nodes but on (a) the number of nodes which replied to RFBs, and (b) the complexity of the queries. Hence, we ended up running three sets of experiments: In the first set, the workload consisted of select-project-join queries with a varying number of 0–29 joins. The relations referenced in the joins had no mirrors and only one partition. This workload was used to test the behavior of the optimization algorithms as the complexity of queries increased. In the second set of experiments we considered 6-way join queries that referenced non-partitioned relations with a varying number of copies (0–29). This experiment measured the behavior of the algorithms in the presence of redundancy. In the last set of experiments, we considered 4-way-join queries that referenced relations with a varying number of 1–9 horizontal partitions. The number of data copies varied from 0–8 to ensure that the algorithms had multiple alternatives for selecting partition data sources. This test measured the behavior of the algorithms as relation data were split and stored (data spreading) in different nodes. This last set of experiments had substantially more complexity than the first two ones, since algorithms had to enumerate all possible ways of splitting and parallelizing joins.

We randomly created 12 path-shaped and 12 star-shaped queries for each possible number of query joins, relation partitions and mirrors (i.e., a total of $(12 + 12) \times (30 + 30 + 9) = 1{,}656$ queries). We batch optimized these queries using all possible algorithms (DP,IDP,IDP-NK,QT,QT-IDP,MARIPOSA). Except for the DP and IDP algorithms, we run the remaining algorithms using both the centralized and the DHT-based (P2P) directory service. Thus, a total of 16,560 query optimizations were run. From the output of our simulations, we calculated the optimization time of the algorithms and the plan cost of the queries, that is, the time required to execute the plans produced as this was estimated



(a) Plan cost of basic algorithms. (b) Breakdown of optimization times.

Fig. 6. Performance of the Query Trading algorithm.

by the algorithms. Optimization time was split into two pieces. The first one was the actual local processing time spent by buyer nodes. The second one (network and seller processing time) was the delay caused by network traffic and remote (sellers') processing. It was not possible to measure these delays separately as they overlap. However, to let readers understand the amount of network resources spent during optimization, we also measured the total number of bytes exchanged through the network. This number is not directly related to the cost (in seconds) of network messages as the latter were dispatched in parallel in both the centralized and the P2P directory service case.

We should emphasize again that the execution times of DP and IDP are not directly comparable to those of the other algorithms, as these are *centralized* implementations, incurring no cost for network message exchange.

5.3 Results

5.3.1 *Number of Joins.* Figures 6(a) and 6(b) summarize the results of the first set of tests measuring algorithms' performance as the number of joins was varied. The first of these figures presents the average plan cost (in seconds) of queries optimized by DP, IDP/IDP-NK (recall that IDP-NK produces the same plans as IDP), Mariposa, and the two instances of QT (plain QT and QT-IDP). The averages are calculated separately over all path- and star-shaped queries optimized in this first set of tests.

As expected, DP and QT produced the same results since no bids were lost and tables had no mirrors. The same holds for IDP and QT-IDP. The Mariposa algorithm made the largest error producing plans that on average required 10% more time than those produced by DP. The reason is that, in its first phase, Mariposa disregarded network costs and delays and thus, failed to find the best joins order for the base tables. The remaining of the algorithms selected the proper joins order, taking into account that delays caused by slow data sources may be masked if joins related with these sources are executed last.

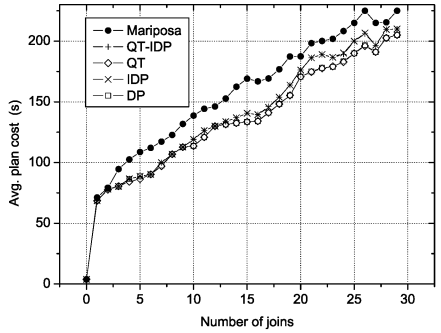
Figure 6(b) presents the average optimization time of all algorithms tested when the directory service used was either centralized or distributed (P2P).

Times are split into two parts: The first one (local processing time) is the time spent by buyer nodes and the second one (network and seller processing time) represents time spent in network communications and sellers processing. The figure shows that from the algorithms suitable for autonomous distributed environments, the IDP-NK is clearly the slowest one, followed by QT, QT-IDP and Mariposa. IDP is the fastest of all but is a centralized algorithm. When a centralized directory was used, the network and seller processing times of all algorithms were small compared to their respective total optimization times. This did not also hold when the distributed directory was used. In that case, the network and seller processing delays were substantial and severely affected total optimization times. It is worth noticing that a large portion of the optimization time of the IDP-NK and P2P IDP-NK algorithms was not due to network delays but due to local processing. This is because the delayed pruning of these algorithms increased their memory requirements slowing down the structures held by our dynamic algorithm implementation.

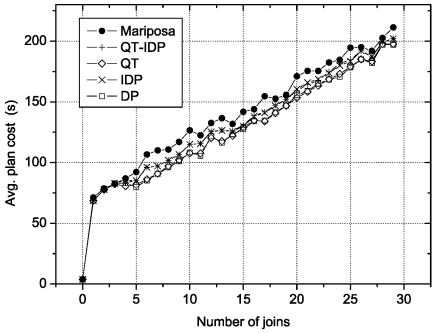
Figures 7(a) and 7(b) display the average plan cost vs the number of joins for path and star queries. The DP and QT algorithms always produced better plans than the IDP and QT-IDP, respectively. The latter ones, deviated from the optimal plan (i.e., that of DP) when the number of joins was more than three. Mariposa produced plans that were up to 20% slower than those of DP. Note that contrary to the case of optimizing nondistributed queries, the plan cost of star-queries was smaller than that of the respective path ones. The reason is that star-queries had more chances of pushing multiple joins to the same remote seller node than path queries.

Figures 7(c) and 7(d) presents the average total optimization time in milliseconds for path and star queries separately, as the number of joins of the queries being optimized is varied. The directory service used is the centralized one. The average optimization time of all algorithms depends exponentially on the number of joins, with star-queries times being larger than path-queries ones. The Mariposa, QT-IDP, and QT algorithms have a start-up cost of a single round of message exchanges (approximately 50 ms) caused by their bidding procedure. IDP-NK requires at least one round of message exchanges and, hence, also has a similar start-up cost. For small number of joins, Mariposa is faster than QT-IDP. However, for large queries (more than 13 joins), Mariposa becomes slower to QT-IDP. This is because Mariposa's greedy algorithm replaces leaf operators of the currently best execution plan in such a way that the *OperatorSchedule* algorithm (see Section B.2 of the electronic appendix) is invoked with the maximum degree of complexity, as the stored sets T and N are invalidated. Lastly, for star-queries with more than 9 joins, our implementation of QT and DP algorithms exceed the soft memory-usage limit (see Section B.3 of the electronic appendix) and fell back to a variation of the QT-IDP and IDP, respectively. This does not seem to harm the quality of the execution plans produced but does affect the optimization times of DP and QT. This is visible in Figure 7(d) where the slope of DP and QT lines is reduced when the number of joins is equal or more than 9.

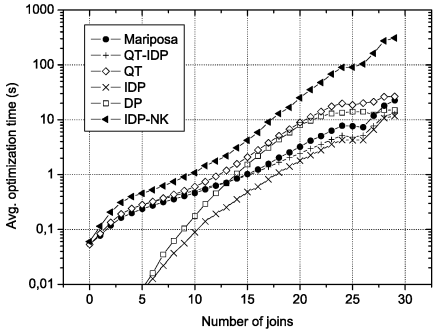
Figure 7(e) displays the additional time required to optimize path queries when the distributed algorithms used the DHT-based directory service instead



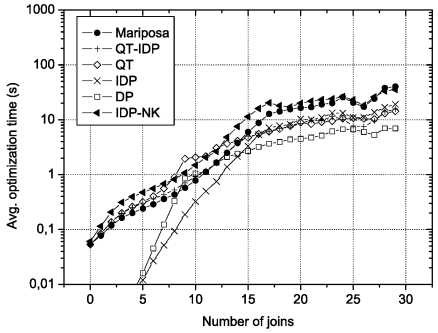
(a) Plan cost of path-queries vs number of joins.



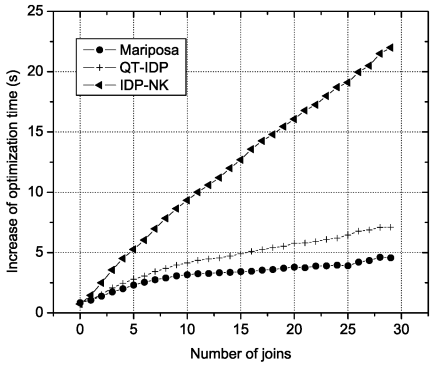
(b) Plan cost of star-queries vs number of joins.



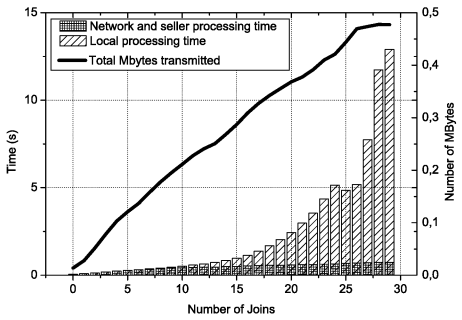
(c) Optimization cost using a centralized directory vs number of joins (path queries).



(d) Optimization cost using centralized directory vs number of joins (star queries).

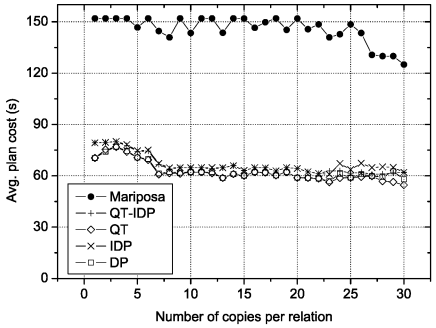


(e) Increase of optimization time when using the distributed directory instead of the centralized one.

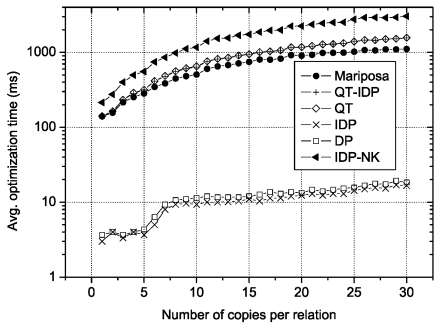


(f) Breakdown of optimization cost of plain QT algorithm vs number of joins.

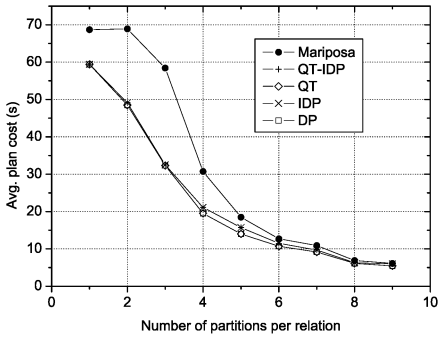
Fig. 7. Performance of the query trading algorithm vs number of joins.



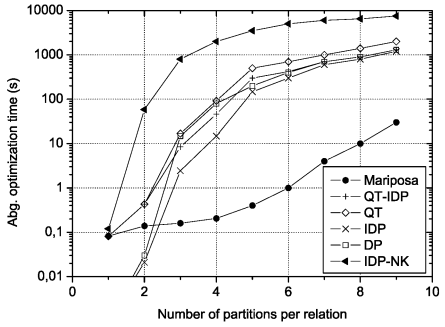
(a) Plan cost vs number of mirrors per partition.



(b) Optimization cost vs number of mirrors per partition.



(c) Plan cost vs number of horizontal partitions.



(d) Optimization cost vs number of horizontal partitions.

Fig. 8. Performance of QT algorithm (path queries with centralized directory).

of the centralized one. As expected, in all cases, this additional time grows almost linearly with the number of joins involved in the query, as more messages are required to get the relevant information for the extra tables. Mariposa has the lowest rate of increase with QT-IDP following up closely, since the size (in bytes) of the messages of the latter algorithm are larger than those of the former. The rate of optimization-time increase of IDP-NK is by far the worst, since the number of message-exchange rounds of IDP-NK linearly depends on the number of joins. This is in contrast to QT-IDP and Mariposa, which completed the message exchanges in a single round for all number of joins tested.

Figure 7(f) examines the average local processing time, the network and seller processing time, and the total number of MBytes transmitted thought the network by the QT algorithm for path queries with the centralized directory service. This figure shows that the volume of exchanged information and nonlocal delays almost linearly depend on the number of joins, whereas the local processing time exponentially depends on the number of joins.

5.3.2 *Data Mirroring.* Figures 8(a) and 8(b) summarize the results of the second set of experiments, where the level of data mirroring is varied. The

figures concern path queries with centralized directory. We also run experiments with star queries and the distributed directory but there were no significant differences. Data mirroring substantially increases the complexity of all algorithms as they have to select the nodes allowing for the best parallelization/pipelining of join (and union) operators. The best plans were produced by DP and QT, though IDP and QT-IDP were not affected by redundancy and also produced almost optimal plans. Figure 8(c) shows that Mariposa produced plans that were more than two times slower than the ones produced by the remaining algorithms, yet, as Figure 8(d) displays, it completed optimization only marginally faster than QT, QT-IDP and IDP-NK.

The IDP-NK algorithm was by far the slowest of all algorithms, followed by QT and QT-IDP. Mariposa was faster than QT and QT-IDP but, as expected, slower than IDP and DP, since the results of the last two algorithms include no cost of network message traffic.

5.3.3 *Data Partitioning.* Figures 8(c) and 8(d) summarize the results of the last set of experiments, which evaluate the algorithms as the partitioning and spreading of information varied. Similarly to the previous set of experiments, the figures concern path queries with centralized directory. Our findings also hold for star-queries and distributed directory services.

The first figure shows that QT and QT-IDP took advantage of the increased possibilities for join splitting and operators parallelism and produced plans of smaller cost as the number of partitions was increased. DP and IDP produced marginally better plans than QT and QT-IDP, respectively, because they enumerated all possible ways of splitting query joins, whereas the query trading algorithms considered only those plans that were possible from the offers received. Mariposa was affected by the level of data partitioning more than the other algorithms and initially produced plans that were up to 90% slower than those of DP. The reason is that as the number of partitions was increased, the chances that some remote partitions had some useful remote indices (which were disregarded by Mariposa as each partition had different indices) were increased. Furthermore, the degree of join splitting and operators parallelism is statically determined by the database administrator and not by the Mariposa algorithm itself. Although, we had configured Mariposa to produce plans of maximum degree of parallelism, the other algorithms automatically choose plans with less degree of parallelism but with better pipelining possibilities, and thus managed to produce plans that were better than those of Mariposa.

Data partitioning substantially increases the search space of the dynamic programming algorithm. For more than three partitions per table, all algorithms, except Mariposa, encountered soft memory overflows (see Section B.3 of the electronic appendix) and fell-back to using the IDP(k,m) heuristic with different but very small values of the k and m parameters. This is visible in Figure 8(a) as for large values of number of partitions, all algorithms produced plans of similar cost.

Figure 8(d) shows the execution time of all algorithms tested. Except for Mariposa, the remaining algorithms spent a lot of time trying to find the optimal way of horizontally splitting query joins. IDP-NK, as expected, was the slowest

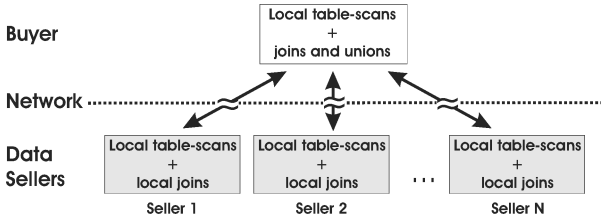


Fig. 9. Processing tasks allocation by the QT algorithm.

of all algorithms whereas, Mariposa was by far the fastest. In fact, in some cases, Mariposa was approximately three orders of magnitude faster than QT. Note that Mariposa optimization time appears to exponentially depend on the number of partitions. This is because the values tested are small. The true complexity of Mariposa was polynomial.

6. TRADING OF PROCESSING TASKS

In distributed query optimization, an important factor affecting the overall performance of distributed execution plans is the selection of nodes that will actually process the data. Figure 9 shows the way processing tasks are assigned by QT, as presented in the previous sections. Note that processing is performed in either the sellers (of query answers) or the buyer, which constitutes a two-level approach. Sellers only process data that is locally available, while the buyer performs all left-over processing on the data received from the sellers. These restrictions on where processing may occur reduce the search space of the buyer, helping query trading to complete in reasonable time. Nevertheless, they may lead to nonoptimal plans, especially in cases where the buyer is overloaded or the network connection between the buyer and the sellers is slow.

In the remaining of this section, we extend the QT algorithm so that it can handle trading of pure processing tasks, that is, assigning the execution of a plan's operators to nodes that do not have all the data required available locally. As we show in Section 6.2, this enables our technique to produce better execution plans at the expense of additional optimization time.

6.1 Algorithms

In general, to trade processing tasks (pt-trade, for short) within a query optimization algorithm, the buyer broadcasts to candidate sellers an RFB for *specific* operators/subqueries that appear in the query being optimized. Such an RFB includes all needed information about the input data of the operators/subquery, such as their size and data distribution, since this data may not be local to the sellers. The buyer must have this information from earlier steps of the algorithm, while the sellers can use it to estimate the execution cost of (some or all of) the operators and make the appropriate offers. The buyer receives these offers, adds to the costs stated by the sellers the corresponding costs to transfer the necessary data to them, and based on the results, chooses the best offers for consideration in the next step of the algorithm. Note that the above exchange still respects the autonomy of all nodes; compared to QT, all

Buyer-side algorithm	Sellers-side algorithm
...	...
B8. Identify all operators of plan P_*. For each of these operators, request processing bids (processing-RFB) from remote nodes.	
	S4. For each processing operator, estimate the cost (and more generally the properties) of evaluating it locally. The estimations assume that all needed data are available at the seller site.
B9. For each operator, select the K best offers. In addition, select all offers coming from interesting sites.	S5. Using the trading framework, make offers for the operators evaluated at step S4.
B10. Using P_* and the processing offers selected in step B9. find a new optimal distributed plan P'_*	
B11. Inform the selling-nodes, which offered queries and processing used in plan P'_*, to execute these queries/operators.	

Fig. 10. Query trading with final step of pt-trading (QTPT).

additional information exchanged becomes available during the natural progression of the algorithm and requires no revelation of internal information.

We have identified three different extensions of QT to incorporate pt-trading in it. These are analyzed below.

Query Trading Followed by Final Step of pt-Trading (QTPT). This is the simplest way of extending QT with pt-trading. It only requires modification of the buyer predicate analyzer module (see Figure 3) and works in two steps. First, it runs the normal query trading algorithm to find an initial distributed query execution plan and then, it runs a single pt-trading round to assign processing of plan operators to remote nodes that would otherwise be processed by the buyer after QT.

QTPT is different from QT (Figure 2) after step B7. Figure 10 shows only the steps that are modified. More specifically, in step B8, instead of terminating, QTPT locates all processing operators of the optimal plan P_* built by QT and asks for processing bids from remote nodes. Seller nodes estimate the cost (and other relevant properties) of evaluating these operators locally using the cost model and knowledge that they have about their locally available resources (step S4). These estimations assume that the input data of these operators are available locally at the seller site. Subsequently, in steps B9 and S5, a traditional trading procedure helps the buyer to decide which processing offers should be accepted.

An important difference from QT is that, instead of selecting a single offer per operator, the buyer selects the best K offers, where K is an administrator-provided constant. The reason for keeping K offers is that, until step B10, we do not know the exact origin of the input data streams of each operator. For instance, for a five-way join, plan P_* designates the exact origin of the raw table-scans but not the origin of the inputs of the intermediate joins (these are most likely "temporarily" assigned to the buyer node in P_*). Increasing

Buyer-side algorithm	Sellers-side algorithm
...	...
B4a. Identify all operators of plans P_m. For each of these operators, request processing bids (processing-RFB) from remote nodes.	
	S4. For each processing operator, estimate the cost (and more generally the properties) of evaluating it locally. The estimations assume that all needed data are available at the seller site.
B4b. For each operator, select the K best offers. In addition, select all offers coming from interesting sites.	S5. Using the trading framework, make offers for the operators evaluated at step S4.
B4c. Using plans P_m and the processing offers selected in step B6. find new optimal distributed plans P'_m	
...	...

Fig. 11. Iterative query-answer and processing-task trading (IQPT).

K produces better execution plans but exponentially increases optimization time as well. In the experiments presented in this section, K equals either 3 or half the number of joins of the query being optimized, whichever is larger. In addition to the K best offers and in accordance with Kossmann [2000], the buyer also unconditionally accepts offers from all interesting sites. These are the seller nodes providing query-answers in P_*. After selecting the pt-offers, the buyer DP algorithm is run for the last time starting from the table-scans of P_* and taking these offers into account. The result is a new execution plan P'_*, which is executed in step B11.

The QTPT algorithm does not substantially increases the optimization time of QT but has the potential of substantially improving the plans produced (due to wider distribution of processing tasks). However, it is a two-step optimization procedure, still leaving space for further improvements of the produced plans.

Iterative Query and Process Trading (IQPT). The drawback of QTPT is that, in step B7, the algorithm has already decided on the nodes that will provide the data (original relations or partial subquery results), taking into account not the entire network of nodes but only the buyers and data-sellers. This problem is partially solved by the IQPT algorithm. The algorithm proceeds like QT, but query-answer trading cycles are interchanged with the pt-trading cycles that assign processing tasks to remote nodes. That is, between steps B4 and B5 of QT (Figure 2) a pt-trading cycle is inserted (Figure 11). Each pt-trading cycle is handled in a similar way as in QTPT. The distributed plans P'_m produced are fed (steps B5 and B6 of Figure 2) into the buyer predicate analyzer, which is responsible for finding additional queries whose trading could further improve the plan (in the next iteration of the algorithm).

The IQPT algorithm is still a two-step optimization method (albeit iterative), since in step B5, plans P_m have already fixed the query-answer providers. The difference between QTPT and IQPT is that, in the latter, pt-trading does affect the potential join and union order of distributed plans P'_m, which are used by

Buyer-side algorithm	**Sellers-side algorithm**
B0. Initialization, set $Q = \{\{q, C_i\}\}$, $PC = \emptyset$ B1. Make estimations of the values of the queries in set Q, and processing operators in set PC using a trading strategy. B2. Request offers for the queries in set Q and the processing operators in set PC	S1. For each query q in set Q do the following: S2.1. Find sub-queries q_k of q that can be answered locally. S2.2. Estimate the cost c_k of each of these sub-queries q_k. S2.3. Find other (sub-)queries that may be of some help to the buyer node. S3. For each processing operator in set PC, estimate the cost (and more generally the properties) of evaluating it locally. The estimations assume that all needed data are available at the seller site.
B3. Select the best offers $\{q_i, c_i\}$ using **one of the three** methods (bidding, auction, bargaining) of the query trading framework. Additionally, select the K best processing offers and all processing-task offers coming from interesting sites. After selecting the processing-task offers, **select additional query-answers using the SQPT-qans() algorithm**.	S4. Using the query trading framework, make offers and try to sell some of the subqueries of step S2.2 and S2.3 and processing operators of step S3.
B4. Using the selected query and processing offers, find possible execution plans P_m and their estimated cost C_m B5. Find possible sub-queries q_e and their estimated cost c_e that, if available, could be used in step B4. B6. Update set Q with subqueries $\{q_e, c_e\}$. B7. Locate all processing operators in queries P_m and in queries q_e and insert them in set PC. B8. if step B6 modified set Q or step B7 modified set PC, then go to step B1. B9. Inform the selling-nodes, which offered queries and processing used in plan P_*, to execute these queries/operators.	

Fig. 12. Simultaneous query-answer and processing-task trading (SQPT).

the buyer predicate analyzer. Thus, we expect the latter to produce queries (for use in the next iteration of the algorithm) that are better suited for the globally available processing resources. Overall, IQPT is more complex than QT and QTPT but finds better plans, although its two-phase nature still leaves space for improvement.

Simultaneous Query and Processing Trading (SQPT). This algorithm (Figure 12) tries to overcome the two-phase nature of the previous algorithms by

embedding pt-trading into the heart of QT (modification of the buyer predicate analyzer again) and simultaneously requesting bids for both query answers and processing tasks. Hence, in addition to keeping a set of queries Q whose answers might be worth purchasing from remote nodes, it also keeps a set PC of operator-processing tasks that it tries to purchase from remote nodes in every iteration of the algorithm. The SQPT algorithm starts with an empty PC set (step B0), which means that in the first iteration, only query answers are traded. At the end of the first iteration, PC is updated with all processing tasks that are present in the set P_m of currently best plans and in the set q_e of subqueries that will be requested in the next iteration of the algorithm (step B7). In the second and subsequent iterations, SQPT trades query answers (set Q) and processing-tasks (set PC) on an equal basis (step B2–B3). Accordingly, the buyer DP algorithm (step B4) takes into account offers produced for both sets.

In step B3, selection among query-answer and processing-task offers is handled as follows. First, similarly to QT, the best query-answer offers are accepted, that is, those offers that together with the network cost to move the data to the buyer node have the overall least cost. Processing-task offers are then selected in a way similar to QTPT and IQPT. Finally, except for the first iteration of SQPT, the following algorithm is run to select additional query-answer offers that minimize the network cost to nodes that have made the best relevant pt-offers:

SQPT-qans algorithm

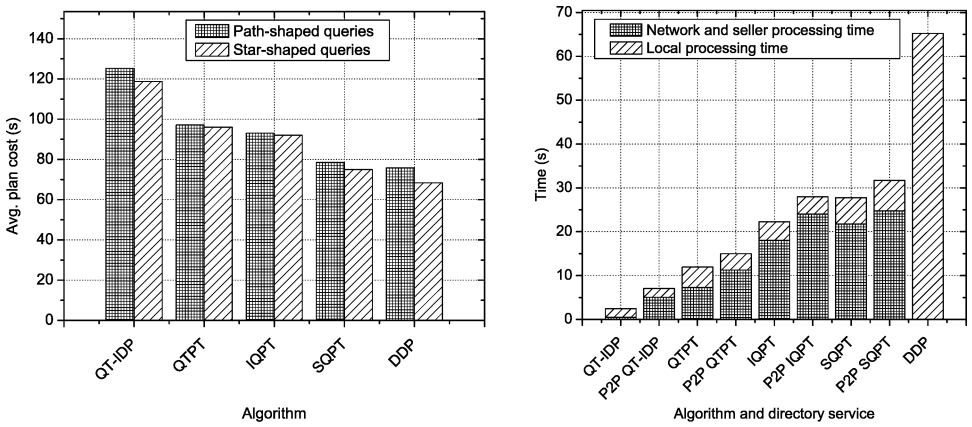
```
1  Set N = ∅.
2  For each query q_e in set Q do
3     Find all selected processing-task offers that take as input this query q_e. Insert
      the nodes that made these offers into set N.
4     For each node n in set N do
5        Accept the relevant query-answer offer that together with the network cost
         to move the data to node n has the overall least cost.
6     End for
7  End for
```

As mentioned above, the first iteration of SQPT is special in that it does not include any pt-trading. The reason is lack of two pieces of information for the buyer. First, the buyer has not built any execution plan P_m and thus does not know which operators will be needed for answering the query being optimized to ask for bids to process them. Second, the buyer has no information on any data in order to propagate it to potential sellers so that they may produce their offers.

Overall, SQPT has substantially higher complexity than the previous algorithms, since its buyer DP algorithm simultaneously examines both query-answer and processing-task offers. On the other hand, it produces better plans.

6.2 Experiments

In order to test the performance of the three previous algorithms, we have used our network simulator to run a new set of experiments. The parameters of the simulated network and data sets are the same as those of Section 5.1. As a basis for comparison, we have used the Distributed Dynamic Programming (DDP) algorithm presented in Kossmann [2000]. This algorithm is similar to



(a) Average plan cost. (b) Average cost of optimization.

Fig. 13. Performance of query and process trading algorithms.

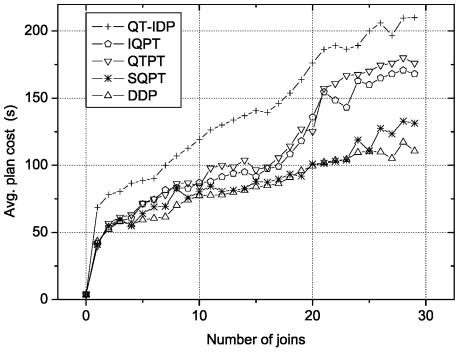
classical IDP, but each join operator is examined in all possible interesting nodes using all possible mirrors of the outer and inner relations. Interesting nodes are those having data relevant to the query. A plan is pruned only when the cost of the plan plus the shipping cost of moving the results of this plan to all other interesting nodes is dominated by another plans (when moved to the same node) producing the same results.

We have run the same three sets of experiments using the same randomly created set of queries as in Section 5.1, and measured the average optimization time and plan execution time for DDP and each of the pt-trading algorithms using either a centralized directory service or a distributed one (11,564 query optimizations). In all experiments and all algorithms, the IDP(3,5) algorithm was enabled since it reduced their optimization time without substantially hurting the performance of the execution plans produced.

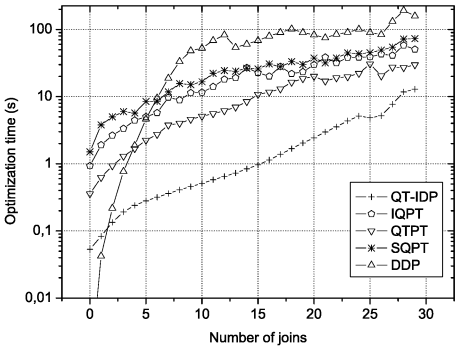
As in Section 5.1, the execution time of DDP is not directly comparable to QT,QT-IDP,IDP-NK and Mariposa, as it does not incur any cost of network message exchanges. For comparison reasons, in the graphs presented in the next section, we include the respective results of Section 5.3 obtained by running QT-IDP. We differ the comparison of the pt-trading algorithms to IDP-NK and Mariposa till Section 9.

6.3 The Results

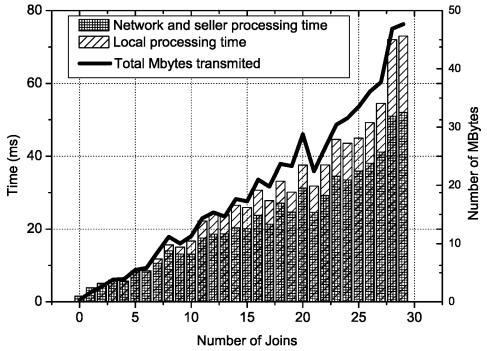
6.3.1 *Number of Joins.* Figure 13(a) presents the average execution cost of plans produced by DDP, QT-IDP and the three QT enhancements with pt-trading, for both path-queries and star-queries. The results confirm our intuition that the pt-trading algorithms would substantially improve the plans produced by QT-IDP. Figure 14(a) further analyzes the performance of the algorithms as the number of joins is varied for path-shaped queries. The results obtained for star-queries are similar. This figure shows that pt-trading algorithms always produce better plans than QT-IDP irrespectively of the complexity of queries being optimized. However, as the number of joins is



(a) Plan cost vs number of joins.



(b) Cost of optimization vs number of joins.



(c) Breakdown of optimization cost of SQPT algorithm vs number of joins (path queries with centralized directory).

Fig. 14. Performance of query and process trading algorithms vs number of joins.

increased, the average quality of plans produced by QT-IDP, QTPT and IQPT deviates from that of DDP and only SQPT manages to perform close to DDP.

The average optimization times of path-queries for the previous algorithms are presented in Figure 13(b), either with a centralized directory or with a distributed (P2P) one. Interestingly, in both cases, the pt-trading algorithms spend a lot of their time idle, waiting for replies to their RFBs concerning queries and processing tasks. This is why the network and seller processing delay times are a large fraction of the total optimization time of pt-algorithms. The relative difference of the total optimization time between the centralized directory and the distributed one is rather small for pt-algorithms compared to that of QT.

Figure 14(b) analyzes the optimization time vs number of joins of path queries with centralized directory. The execution time of the algorithms exponentially depends on the number of joins. As expected, the QT-IDP algorithm has a start-up cost of a single round of message exchanges (approximately 50 ms) caused by the bidding procedure. The QTPT algorithm has a start-up cost of two rounds of message exchanges (and two runs of the buyer DP

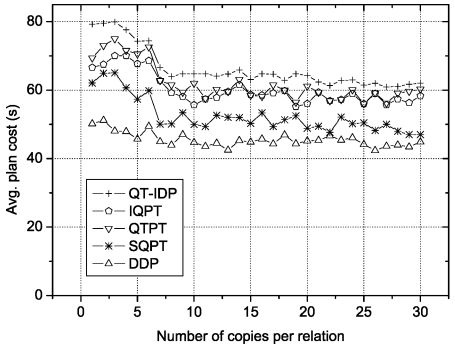
algorithm). The IQPT algorithm has a start-up cost of four rounds of message exchanges (and four runs of the buyer DP algorithm) since it runs at least two iterations. Finally, the SQPT has a start up cost of two rounds of message exchanges and two runs of the buyer DP algorithm. Its substantially higher cost is due to the much higher complexity of its DP module algorithm, which is higher than that of the other trading algorithms.

Figure 14(c) further analyzes the behavior of the SQPT algorithm by presenting the average local processing times, the network and seller processing delays, and megabytes transmitted vs the number of joins. The directory service used is the centralized one, and queries are path-shaped. Due to processing parallelization (by seller nodes) and the fact that the k and m parameters of IDP were modified at run-time whenever a soft memory overflow was detected, the optimization time, the network delays and the megabytes transmitted by the SQPT algorithm almost linearly depend on the number of joins. Still, comparing Figure 14(c) to Figure 7(f) we see that the SQPT algorithm transmits a lot more data through the network than plain QT.

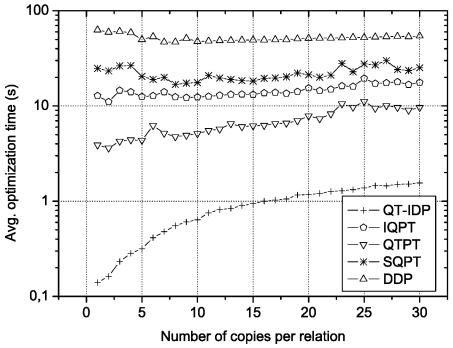
6.3.2 *Data Mirroring.* Figures 15(a) and 15(b) summarize the results of the second set of experiments, where the level of data mirroring is varied. The presented figures concern path queries with a centralized directory. The first figure shows that all trading algorithms benefit from the existence of data mirrors. Initially, as the number of copies is increased all algorithm improve their execution plans by parallelizing join operations and using bushy and hyper-bushy execution plans (see Figure 22(c)). However, for more than 8 copies per relation, all algorithms roughly fail to further improve the produced plans.

As far as optimization cost is concerned (Figure 15(b)), SQPT is the slowest of the trading algorithms followed by IQPT, QTPT, and finally QT-IDP. Optimization time of all pt-trading algorithms is affected by the increase in the number of network message exchanges and the increase in their search space. Fortunately, the bidding procedure discards most offers for the same part of a query received from different seller nodes, keeping only the best K offers (see Section 6.1). Thus, the search space and optimization times of the pt-trading algorithms is kept substantially smaller than those of DDP, which considers all nodes having data relevant to the query.

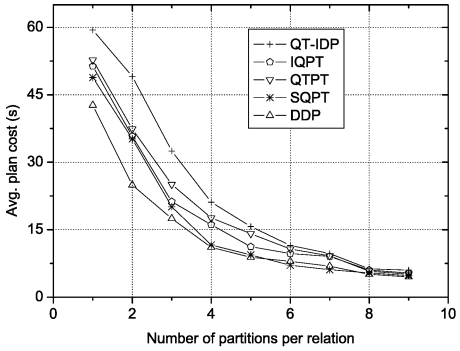
6.3.3 *Data Partitioning.* Figures 15(c) and 15(d) summarize the results of the last set of experiments, where the number of partitions and mirrors is varied. The figures concern path-queries with a centralized directory. All pt-trading algorithms manage to improve the plans produced by plain QT-IDP. However, as the number of partitions is increased, the complexity (and memory) requirements of all algorithms are substantially raised. For values larger than 3 partitions, all algorithms automatically modified their IDP(k, m) parameters to avoid memory overflow (see Section B.3 of the electronic appendix). For more than 5 partitions, all algorithms ended up running using $k = 2, m = 1$, which severely affected the quality of plans produced. This is why for large values of the number of partitions, all algorithms behave similarly. As far as the optimization time is concerned, this increases almost exponentially with the number of relation partitions.



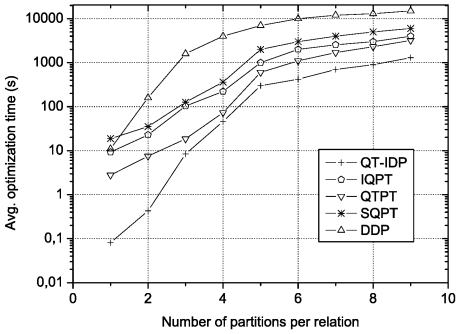
(a) Plan cost vs number of mirrors per partition.



(b) Cost of optimization vs number of mirrors per partition.



(c) plan cost vs number of horizontal partitions.



(d) Cost of optimization vs number of horizontal partitions.

Fig. 15. Performance of query and process trading algorithms.

7. SUBCONTRACTING

7.1 Algorithm

Up to now, we have assumed that all nodes could directly communicate to all other ones. However, this is not always true, especially in networks of autonomous systems. For instance, nodes of autonomous interconnected LANs (except for *gateways*) are typically aware of (and are allowed to directly communicate to) only those nodes residing on the same LAN. Query subcontracting can be used by our trading framework to allow it to work in cases where such nodes' interconnection restrictions apply, or when nodes use different (centralized or distributed) directory services, which hold the same type of information but for different sets of nodes (part of the network). Such *segmentation* of the network interconnection graph or of the directory service information occurs frequently due to security constraints (e.g., firewalls), other policies, or simply due to the physical network topology. Allowing sellers to subcontract parts of queries requested by buyers can also potentially improve the overall quality

of plans produced, as sellers will try to outsource those parts that cannot be handled efficient by them.

Modifying either QT or one of the pt-trading algorithms to support query subcontracting is relatively straightforward. Only the seller-side algorithm of Figure 2 is altered as follows:

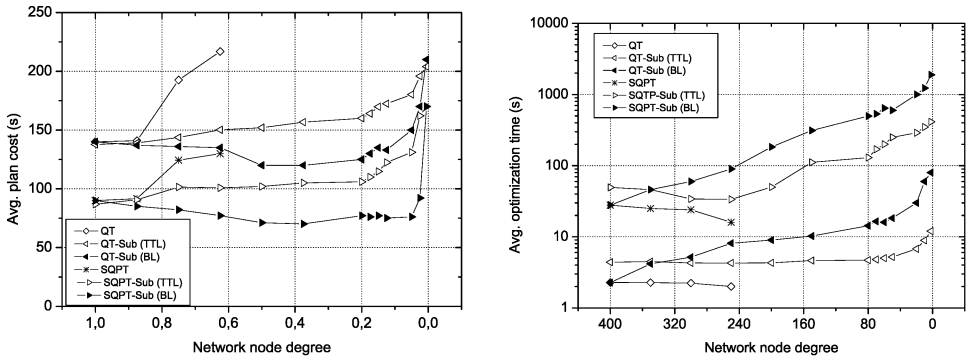
Sellers-side algorithm

S1. For each query q in set Q do the following:
S2.1. If $subcontracting IsAllowed()$ then
S2.2.1. Run QT (or a pt-trading algorithm) to optimize query q.
S2.2.2. For each execution plan p that was not pruned by the DP algorithm of the Buyer Query plan generator of QT (see Section 3.6) do:
S2.2.3.1 Find the part q_p of query q that is evaluated by plan p. The cost c_p of q_p is the cost of plan p as found by the Buyer Query plan generator of step S2.2.1.
S2.2.4. End For.
S2.3. else
S2.4.1. Find subqueries q_k of q that can be answered locally.
S2.4.2. Estimate the cost c_k of each of these subqueries q_k.
S2.5. End if
S2.6. Find other (sub-)queries that may be of some help to the buyer.
S1. End For.
S3. Using the query trading framework, make offers and try to sell some of the sub-queries of steps S2.3.1, S2.4.2 and S2.6.

The above algorithm differs from the original seller algorithm of Figure 2 in steps S2.1–S2.3. For each query q requested by a buyer, a nested query trading (QT or pt-trading) algorithm is run (step S2.2.1) at each buyer-reachable seller node. This algorithm optimizes q using the remotely available data and the seller's local data. The results of this procedure (S2.2.3.1) are then offered to the initial buyer. If insufficient seller-reachable nodes exist, the nested trading algorithm of step S2.2.1 may fail to find a complete execution plan of the whole query q, therefore, sellers make offers (to the initial buyer) for all plans (S.2.2.2.) that were simply not pruned by the Buyer Query plan generator of these nested QT or pt-trading algorithms.

The seller algorithm for subcontracting is recursive, since at step S2.2.1 a query trading algorithm is instantiated which, in turn, recursively makes use of the seller algorithm. This recursion is terminated at step S2.1. with the help of the *subcontractingIsAllowed*() function. We tested two different implementations of this function. The first one is based on a technique borrowed from the area of network routing. It uses a Time To Live (TTL) integer, which is included in RFBs and decremented each time a nested trading algorithm is used. The initial buyer selects a TTL value large enough to ensure that all network nodes are reached though the recursion.

Our second implementation of *subcontractingIsAllowed*() uses a node *black* list, attached in each query RFBs. Each new instance of a nested trading algorithm uses the black list received (by the seller) to send its new RFBs only to those nodes that are not contained in this list. Furthermore, the black lists included in new RFBs produced by this nested trading algorithm are expanded



(a) Plan cost vs network node degree.

(b) Cost of optimization vs network node degree.

Fig. 16. Performance of subcontracting algorithms when network node degree is varied.

to include all seller-reachable nodes, which means that after a number of recursive instantiations of the trading algorithms, all nodes will be blacklisted and recursion will terminate.

7.2 Experiments

To test the performance of our optimization algorithms with subcontracting, we used the same simulation test-bed as before. The relations referenced in these queries had on average 2 copies. We tested the QT and SQPT algorithms with and without subcontracting using either the TTL or the BlackList (BL) mechanism. To avoid out-of-memory problems in the machine used by our simulation program, the number of nodes was reduced to 400 and the number of relations to 1,000, since due to recursion, hundreds of trading algorithms had to be concurrently simulated. The network was generated as a random graph where each node had a degree of d. In our experiments, d varied from 399 (complete graph) to 2.

Figure 16(a) presents the average plan cost as a function of d for 15-way join path-queries. The QT and SQPT algorithms could not always find an execution plan for all queries tested when node degree was less than 300. For $d < 250$, the node degree was too low and very few execution plans could be found by either QT or SQPT. On the other hand, their subcontracting variations successfully found efficient execution plans for all values of d tested.

For the same node degree, the subcontracting versions of the algorithms produce better plans than their respective plain (nonsubcontracting) versions. This is because the former, through subcontracting, are able to assign processing tasks to nondata providing nodes, that is, perform some kind of (additional) process trading. For instance, a seller node without any relevant to a query data can obtain (by subcontraction) the two halves of this query from two different seller nodes, perform the missing operations to construct the whole query and offer it to the initial buyer.

Figure 16(a) also shows that for the same node degree, the SQPT-based algorithms perform better than those based on QT. This is because the latter still

select data sources and assign processing tasks in separately discrete steps, whereas the former do so simultaneously, as they are extensions of SQPT. As d decreases, the quality of plans produced by QT and SQPT deteriorates as fewer candidate sellers are available. The plan cost of queries optimized by QT-TTL and SQPT-TTL also increases as d is reduced, but at a much slower rate. Such increase cannot be avoided since the network costs of plans produced rise due to the increase in the number of network hops required to contact the physical seller nodes of data.

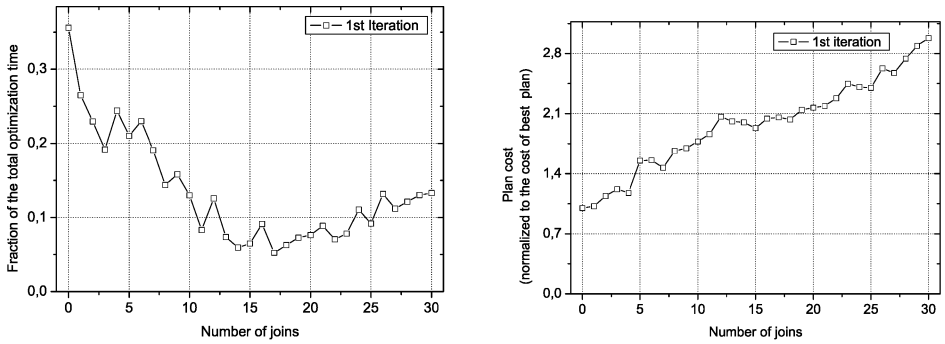
The performance of QT-BL and SQPT-BL is more spectacular than QT-TTL and SQPT-TTL. Plan cost of queries optimized by the blacklist-based algorithms actually improves as d decreases from 399 to 150, as the size of the initial black list of each query is reduced and thus the potential of finding a profitable subcontract is initially increased. The black list ensures that *all* nodes not reachable from the initial buyer node are informed of RFBs. Each of these nodes receives the same RFB from multiple different candidate sellers of the initial (first-level) QT or SQPT algorithm. This is in contract to the TTL mechanism which simply statistically ensures that all nodes receive (once) the RFBs of the initial buyer node. The blacklist-based algorithms consider a substantially larger number of subcontracting possibilities than their respective TTL-based ones and, thus, manage to produce better plans than the latter. For values of d between 150 and 40, QT-BL and SQPT-BL cannot further improve the plans produced and the increase of network costs causes an overall deterioration of plans produced. Finally, for very small values of d, the performance of all subcontracting algorithms collapses due to the substantial increase in the cost of network transfers.

Figure 16(b) presents the optimization time of the algorithms tested for 15-way join path-queries. As expected, the situation is now reversed: For the same node degree, the QT-based algorithms are faster than their respective SQPT-based ones and the subcontracting algorithms are slower than their respective non-subcontracting versions. The optimization time of plain QT and SQPT algorithm is reduced as d is decreased, since the number of sellers contacted is reduced. The QT-BL and SQPT-BL algorithms increase their optimization time as d is decreased, since the number of concurrently running seller-algorithms required to complete optimization grows substantially with that.

The QT-TTL and SQPT-TTL algorithms initially reduce their optimization time as d falls from 399 to 200. For these values of d, the TTL value is constant (2) and thus, as d is decreased, the number of concurrent seller-algorithm instances is decreased. For values of d less than 200, the initial TTL value is starting to increase, causing the number of concurrent seller algorithms to stabilize to approximately 400 instances. This also increases the number of network hops and thus the optimization time is raised as the value of d is reduced from 200 to 2.

8. FALLBACK ALGORITHM

Our main objective when designing the query trading framework was to create algorithms that would respect the autonomy of nodes and produce plans as



(a) Optimization time spent on first iteration of SQPT ($\alpha_1(j)$).

(b) Cost of plans produced in first iteration of SQPT ($\beta_1(j)$).

Fig. 17. Analysis of iterations of SQPT.

close as possible to the optimal (DDP) ones. However, there are cases where query optimization times are large compared to the execution time of plans produced, for example, queries referencing small relations. In such cases, using trading algorithms as presented until now is a problem, especially in interactive environments, where queries are optimized on the fly and the resulting plan is typically used only once. We have studied three possible approaches to reduce optimization time:

(1) reducing the number of iterations of the algorithms,
(2) reducing the extent of partition parallelism exploired by the search component of DP (see Section 3.6),
(3) using a bargaining/auction negotiation protocol, when this is better than bidding.

We examine the first two of these options separately below and consider the use of bargaining/auctions in Section C of the electronic appendix. In all cases, we compare the resulting algorithms to Mariposa. We do not consider IDP-NK since it produces very similar execution plans as QT-IDP, yet, it needs more time to complete query optimization than the latter (see Figures 6, 7, 8). As a case example, we will use the SQPT algorithm; nevertheless, our findings are applicable to all trading algorithms presented in this article.

8.1 Number of Iterations

Since SQPT is an iterative algorithm, it is possible to substantially reduce its optimization time by reducing the number of iterations it goes through. Consider queries requiring more than one iteration to be optimized. Figure 17(a) shows the average fraction of optimization time spent in the first iteration (denoted $\alpha_1(j)$) vs. the number of joins j of the queries. Figure 17(b) presents the corresponding average execution cost of plans produced in the first iteration of SQPT, normalized by the cost of plans produced in the last iteration of the same algorithm (this normalized cost is denoted $\beta_1(j)$). For instance, for queries with 5 joins, the first iteration of SQPT takes approximately 21% of the total

optimization time and produces plans that are on average 55% slower that those of the last iteration. The particular diagrams were derived from the logs of the experiments run in Section 6.3.1 and concern path queries with centralized directory (results for other cases were similar).

Using functions α_1 and β_1, we can heuristically decide whether or not it is worth continuing to the second iteration of SQPT. Let T_1 be the time spent in the first iteration of SQPT for a specific query with j joins, and C_1 be the cost of the best execution plan at the end of this iteration. If we stop optimization at this point, the total cost of optimizing and executing the query will be T_1+C_1. If we continue optimization, then the final cost will be approximately $\frac{T_1}{\alpha_1(j)}+\frac{C_1}{\beta_1(j)}$. Hence, it is worth continuing optimization if and only if

$$\frac{T_1}{\alpha_1(j)}+\frac{C_1}{\beta_1(j)} < T_1+C_1 \iff \frac{T_1}{C_1} < \frac{\alpha_1(j)(\beta_1(j)-1)}{\beta_1(j)(1-\alpha_1(j))} = \lambda_1(j)$$

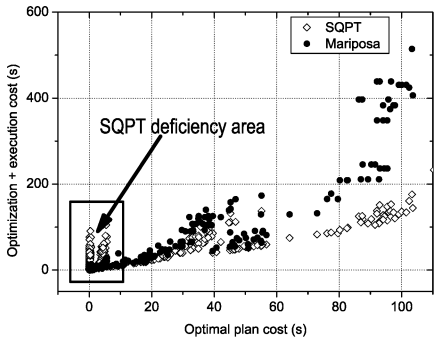
The last fraction is denoted $\lambda_1(j)$. Similarly, we can define α_n, β_n, and λ_n, $n = 1, 2, \ldots, n$, for the n-th iteration of SQPT (α_n would refer to the cummulative optimization time for the first n iterations).

To implement the previous heuristic, step B8 of SQPT (see Figure 12), which is run at the end of each iteration, is modified to test whether equation (1), extended for arbitrary n, holds. If this is not the case, then SQPT is terminated. Fraction $\frac{T_n}{C_n}$ is easy to be calculated at step B8, since both T_n and C_n are known. Function λ_n can be estimated from previous historic data or it can be specified based on system policy.

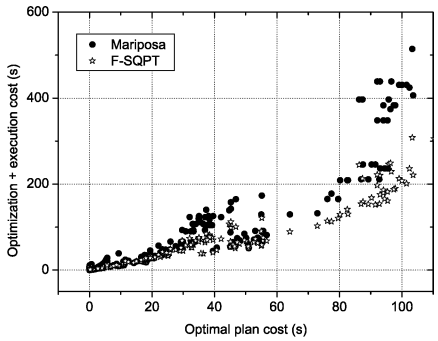
To test the effectiveness of this fall-back strategy, we created a set of 1,000 path queries with 0–29 joins. We created a modified version of SQPT (F-SQPT) which stopped optimization in the first iteration according to the mechanism described above. Function λ_1 was estimated from historic data, so we first use F-SQPT to optimize the first half of the queries (training) and then optimized the remaining 500 queries with SQPT, Mariposa, DDP, and F-SQPT. The results of this experiment are given in the scatter plots of Figure 18 (each point represents an individual query). The x-axis measures the execution costs of queries as estimated by DDP. The y-axis measures the sum of optimization plus execution cost of the queries optimized by SQPT, F-SQPT and Mariposa. Note that in Figure 18(a), SQPT performs better than Mariposa for large queries but is deficient for queries taking only a few seconds to complete. Figure 18(b) shows the behavior of F-SQPT compared to that of Mariposa. Almost always, F-SQPT performs better than Mariposa. Figure 18(c) compares SQPT to F-SQPT. The latter is clearly better for queries with small execution time but the former is better for the remaining ones. The reason is that often F-SQPT incorrectly stops optimization in the first iteration. Hence, SQPT should be further modified to fall back into F-SQPT only when C_1 (or C_n) is low.

8.2 Partition Parallelism

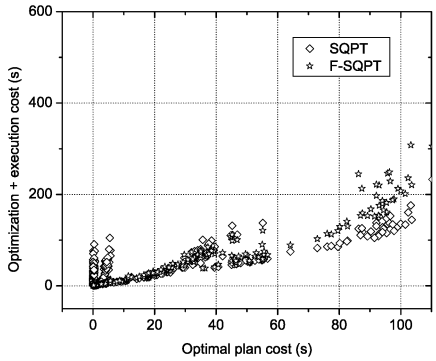
Partition parallelism is a major source of query optimization complexity as the number of ways join and union (of table partitions) operators can be ordered is immense. To obtain a better sense of the impact of such parallelism we used



(a) Comparison of SQPT to Mariposa.



(b) Comparison of F-SQPT to Mariposa

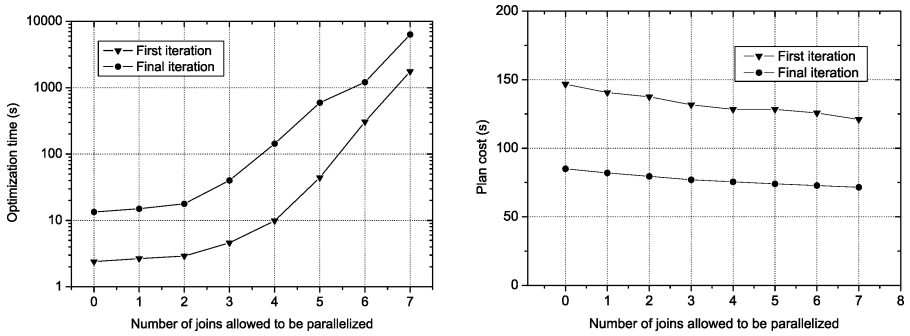


(c) Comparison of SQPT to F-SQPT

Fig. 18. Behavior of F-SQPT algorithm.

SQPT and a centralized directory service to optimize 20 path queries, with 7 joins, where each relation joined had two partitions and two copies per partition. Figure 19 displays the average optimization time and plan execution cost of the first and last iteration of SQPT as a function of the number of relations that are allowed to be executed in a partition-parallel fashion (joins before unions). In particular, for a specific value P in the x-axis, parallel execution of the largest P tables is explored, while the remaining tables are first unioned and then joined as a whole. The results show that the optimization time of SQPT (final iteration) is substantially reduced from approximately 6, 500 seconds (all joins possibly parallelized) down to 13.4 seconds (no parallism), whereas the execution cost of plans produced increases from 71,5 seconds to 85 seconds, respectively. This indicates that reducing the level of partition parallelism examined is a good trade-off between optimization time and plan cost.

Figure 19 also shows that the first and last iteration of SQPT exhibit the same behavior. This indicates that reducing partition parallelism is *orthogonal* to reducing the number of iterations. Hence, the following fall-back strategy of SQPT can be used: After buyer nodes have received all bids from sellers and have selected data sources using the biding procedure, they can estimate the final cost of queries assuming joins are executed locally, without parallelism. The



(a) Optimization time. (b) Plan cost

Fig. 19. Complexity of partition parallelism.

estimated costs together with previous historic data can then be used to decide whether or not the remaining iterations of SQPT should consider a restricted level of partition parallelism. We measure the performance of this fall-back algorithm, together with the use of different negotiation protocols, in Section C.1 of the electronic appendix.

9. DISCUSSION

Distributed query optimization requires extensive knowledge of the current state of network nodes. This includes the location of table partitions, the available indices and materialized views, the network interconnections throughput, the sizing and capabilities of remote nodes, their cost functions and their current workload. In the autonomous environments that we consider, nodes have to ask others for this information, as no node has exact knowledge of all these parameters. Centralized optimization algorithms, such as DP and IDP, must initially contact remote nodes to retrieve this necessary information and, thus, are inappropriate for autonomous environments. Even so, they find the best execution plan of a query by examining several different solutions and assigning processing tasks only to the buyer and data providers (two-level processing-task assignment). The DDP centralized algorithm finds even better plans by assigning processing-tasks to any interesting node. However, the search space is very large and DDP requires large amounts of memory, even when the iterative dynamic programming heuristic is used.

The IDP-NK optimization algorithm is suitable for autonomous environments and produces the same execution plans as IDP does. However, its delayed pruning technique that decouples cost estimations from nodes running the IDP-NK algorithm has the side-effect of increasing the volume of network-exchanged messages and slowing down the enumeration of candidate execution plans due to the large amount of plans kept in memory. A better alternative for cost decoupling is to distribute the optimization procedure, that is, all (remote) nodes should contribute to the optimization process. This ensures that the search space is efficiently divided among many nodes and at the same time, it minimizes the number of messages exchanged during query optimization, as remote nodes need not inform a node running some centralized algorithm of

the cost of each feasible execution plan. The trading algorithms do so by asking candidate sellers to calculate and find the cost of any possible subqueries and processing-tasks that might be useful for the construction of the global plan. Network flooding can be avoided using standard e-commerce techniques such as agent-based architectures, focused addressing, audience restriction, use-based communication charges, and mutual monitoring [Parunak 1987; Smith 1980].

The Mariposa algorithm works in two phases. First it builds the execution plan, disregarding the physical distribution of base relations and then selects the nodes where the plan will be executed using a greedy approach. In this way, Mariposa and any other two-phase algorithm that treats network interconnection delays and data location as less important aspects of optimization produce plans that exhibit unnecessarily high communication costs [Kossmann 2000] and are arbitrarily far from the desired optimum [Papadimitriou and Yannakakis 2001]. Mariposa cannot take full advantage of advanced access methods that may exist in remote nodes if relations are partitioned and have different indices. Finally, it partially violates node autonomy since it requires accurate statistics and information on the physical database design (i.e., existence of indices on the underlying data sources) [Deshpande and Hellerstein 2002]. This information is used in the first phase of optimization, to improve the quality of execution plans produced.

Query trading algorithms are the ones that better respect the autonomy of remote nodes and protect their privacy. They treat them as true black boxes and run having the minimum possible information on them, apart from what is implied by their bids. They do not need or make any assumptions on the state of remote nodes (e.g., their CPU resources, etc.), not even on the data model that they actually use internally. The only piece of information required is their logical schema, which is exposed through a directory service.

Figure 20 shows a scatter diagram of the algorithms presented in this article. This diagram shows the average plan cost vs the average optimization time for nonpartitioned, nonmirrored, select-join-project path queries with 0-29 joins. The directory service used is the centralized one. The network graph is fully connected. The same diagram for star-queries or mirrored or partitioned relations differs in that the pt-trading and subcontracting algorithms have larger optimization times and Mariposa produces less efficient plans. From left to right, there are four areas. In the left area are the algorithms (DDP, SQPT-TTL, SQPT) producing the best execution plans, that is, those that simultaneously consider the allocation of data and processing without any constrain on the selection of nodes. Next are the algorithms (QTPT, IQPT,QT-TTL) attempting to select data-sources and distribute processing to non-data-providing nodes in separate steps. The third area includes all algorithms (DP,IDP,IDP-NK,QT, QT-IDP) that restrict assignment of processing tasks to nodes that provide data. Finally, in the right area of the diagram is Mariposa which works in two phases (i.e., local and then distributed optimization). The F-SQPT algorithm is not displayed in the scatter diagram since its exact position depends on the sum of optimization and execution times. If the average execution time is large compared to optimization time, like in the case of most of the experiments that we run in Sections 5.1, 6.2 and 7.2, then F-SQPT will be near SQPT. Otherwise,

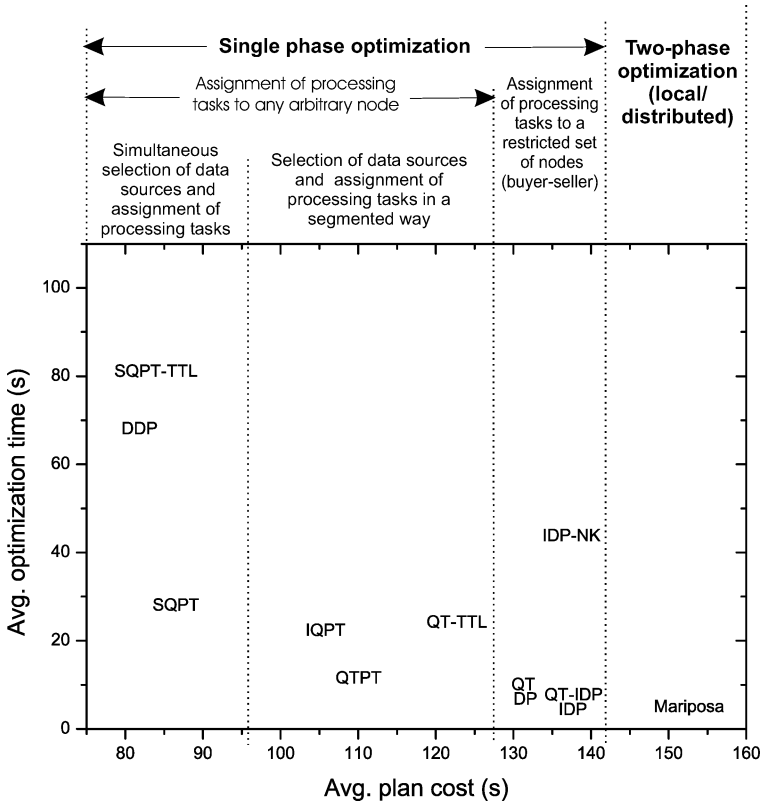


Fig. 20. Optimization time vs plan cost.

Table II. Optimization Algorithms for Networks of Autonomous Database Systems

Small queries	Medium-size queries	Large queries
Mariposa, SQPT with fall-back	QT-IDP, QTPT, IQPT, SQPT	SQPT

F-SQPT will be near QT-IDP. The optimization time and the sum of optimization and execution time of F-SQPT is always between those of QT-IDP and SQPT. The Black-list-based subcontracting algorithms are not displayed because for connected network graphs, they perform similarly to their respective non-subcontracting versions.

Considering the overall trade off between plan execution cost and optimization time as indicated in Figure 20, SQPT (without subcontracting) appears to offer the best trade-off. In interactive environments were possibly small queries may be evaluated, F-SQPT with bargaining (see electronic appendix) and restriction on the level of partition parallelism is the best choice as it keeps the query evaluation time (optimization plus execution time) small. Finally, subcontracting algorithms should be used when buyer has no direct connection with all candidate sellers.

A different perspective is offered by Table II, where the best algorithm choices are shown as a function of query size. Although the fastest of all algorithms in

general (recall that DP, IDP and DDP are centralized algorithms and do not include any network costs), Mariposa produces the worst execution plans. Thus, it should be preferred only for ultrasmall queries (i.e., queries with optimization + execution cost less that a second). Another viable solution for small queries is SQPT with a fall-back mechanism, such as the F-SQPT with bargaining. The QT-IDP and pt-trading algorithms are excellent choices for medium-sized queries, whereas for very large queries, the SQPT algorithm should be preferred.

10. CONCLUSIONS AND FUTURE WORK

In this article, we have discussed an approach to distributed query optimization in a network of autonomous database systems based on a framework for trading query answers and processing tasks. We have elaborated on its main components, explored several algorithmic options and policies, and examined the performance characteristics of the various alternatives. Our algorithms can be used in any large federation of cooperating DBMSs, hosted in a large intranet or the Internet. Through several experimental results, we have showed that these algorithms are scalable and produce plans that offer better tradeoffs between query optimization time and resulting plan execution cost than other existing optimization algorithms. Furthermore, these algorithms require less knowledge about the nodes in the network than other approaches, making them more appropriate for autonomous systems.

To the best of our knowledge, the only other algorithms directly comparable and having similar objectives to query trading is Mariposa and IDP-NK. Our framework improves on most aspects of these algorithms: It requires less global knowledge, produces better execution plans, and it is the only one that runs progressively enabling users to fine tune the query optimization and execution time.

There are several directions that we intend to pursue in our future work. First, we plan to explore the use of *contracts* to model the notion of adaptive/dynamic query optimization and compare the resulting method to other algorithms that exist for the problem. Second, we would like to investigate several theoretical issues with respect to our techniques, for example, whether or not the resulting plans are Pareto optimal, whether or not the algorithms lead to Nash equilibrium, and others. Third, we intend to work on other forms of query-answer valuations (e.g., data completeness or freshness, in isolation or combination) and examine the consequences on our current approaches. Finally, we will explore competitive environments as well.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

BICHLER, M., KAUKAL, M., AND SEGEV, A. 1999. Multi-attribute auctions for electronic procurement. In *Proceedings of the 1st IBM IAC Workshop on Internet Based Negotiation Technologies* (Yorktown Heights, NY).

COLLINS, J., TSVETOVAT, M., SUNDARESWARA, R., VAN TONDER, J., GINI, M. L., AND MOBASHER, B. 1999. Evaluating risk: Flexibility and feasibility in multi-agent contracting. In *Proceedings of the 3rd Annual Conference on Autonomous Agents* (Seattle, WA). ACM, New York.

CONITZER, V. AND SANDHOLM, T. 2003. Complexity results about nash equilibria. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann, San Francisco, CA.

DESHPANDE, A. AND HELLERSTEIN, J. M. 2002. Decoupled query optimization for federated database systems. In *Proceedings of the 18th International Conference of Data Engineering* (Son Jose, CA). IEEE Computer Society, Los Alamitos, CA, 716–732.

FRANKLIN, M. J., JÓNSSON, B. T., AND KOSSMANN, D. 1996. Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Queb. Canada, June 4–6), H. V. Jagadish and I. S. Mumick, Eds. ACM, New York, 149–160.

GANGULY, S., HASAN, W., AND KRISHNAMURTHY, R. 1992. Query optimization for parallel execution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, M. Stonebraker, Ed. ACM, New York, 9–18.

GRAVELLE, H. AND REES, R. 2004. *Microeconomics* (3rd ed). Pearson Education, England.

HALEVY, A. Y. 2001. Answering queries using views: A survey. *VLDB J. 10,* 4, 270–294.

IOANNIDIS, Y. E. AND KANG, Y. C. 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM SIGMOD Conference* (Atlantic City, NJ). H. Garcia-Molina and H. V. Jagadish, Eds. ACM, New York, 312–321.

IOANNIDIS, Y. E., NG, R. T., SHIM, K., AND SELLIS, T. K. 1997. Parametric query optimization. *VLDB J. 6,* 2, 132–151.

KAGEL, J. H. 1995. Auctions: A survey of experimental research. In *The Handbook of Experimental Economics*, J. H. Kagel and A. E. Roth, Eds. Princeton University Press, Princeton, NJ.

KOSSMANN, D. 2000. The state of the art in distributed query processing. *ACM Comput. Surv. 34,* 4 (Sept.), 422–469.

KOSSMANN, D. AND STOCKER, K. 2000. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. Datab. Syst. 25,* 1, 43–82.

KRAUS, S. 2001. *Strategic Negotiation in Multiagent Environments (Intelligent Robotics and Autonomous Agents)*. Bradford Book, Cambridge, MA, USA.

LIU, B. AND RUNDENSTEINER, E. 2005. Revisiting the role of pipelined parallelism in multi-join query processing. In *Proceedings of the 31st International Conference on VLDB*. VLDB Endowment, Trondheim, Norway, 829–840.

MARIPOSA. 2002. *Mariposa Distributed Database Management Systems, User's Manual*. Mariposa, Available at `http://s2k-ftp.cs.berkeley.edu:8000/mariposa/ /src/alpha-1/mariposa-manual.pdf`.

MAS-COLELL, A., WHINSTON, M. D., AND GREEN, J. R. 1995. *Microeconomic Theory*. Oxford University Press, New York.

NAVAS, J. C. AND WYNBLATT, M. 2001. The network is the database: Data management for highly distributed systems. In *Proceedings of the ACM SIGMOD'01 Conference* (Santa Barbara, CA). ACM, New York.

OGSTON, E. AND VASSILIADIS, S. 2002. A peer-to-peer agent auction. In *1st International Joint Conference on Autonomous Agents and Multi-Agent Systems* (Bologna, Italy). ACM, New York.

PAPADIMITRIOU, C. H. AND YANNAKAKIS, M. 2001. Multiobjective query optimization. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on PODS* (Santa Barbara, CA, May 21–23). ACM, New York.

PARUNAK, H. V. D. 1987. Manufacturing experience with the contract net. In *Distributed Artificial Intelligence, Research Notes in Artificial Intelligence*, M. N. Huhns, Ed. vol. 1. Pitman, London, England, 285–310.

PENTARIS, F. AND IOANNIDIS, Y. E. 2004. Distributed query optimization by query trading. In *EDBT*, E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, Eds. Lecture Notes in Computer Science, vol. 2992. Springer-Verlag, New York, 532–550.

POTTINGER, R. AND LEVY, A. Y. 2000. A scalable algorithm for answering queries using views. In *Proceedings of the 26th International Conference on VLDB*, (Cairo, Egypt, Sept. 10–14), A. E.

Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan-Kaufmann, San Francisco, CA, 484–495.

ROSENCHEIN, J. S. AND ZLOTKIN, G. 1994. *Rules of Encounter : Designing conventions for automated negotiation among computers*. The MIT Press series in artificial intelligence, Cambridge, MA.

SANDHOLM, T. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artif. Intell. 135*, 1–54.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, MA), ACM, New York, 22–34.

SMITH, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput. 29,* 12 (Dec.), 1104–1113.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, New York, 149–160.

STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. 1994. Mariposa: A new architecture for distributed data. In *Proceedings of the 10th International Conference on Data Engineering* (Houston, TX, Feb. 14–18). IEEE Computer Society Press, Los Alamitos, CA, 54–65.

STONEBRAKER, M., AOKI, P. M., LITWIN, W., PFELLER, A., SAH, A., SIDELL, J., STAELIN, C., AND YU, A. 1996. Mariposa: A wide-area distributed database system. *VLDB J. 5,* 1, 48–63.

SU, S. Y., HUANG, C., HAMMER, J., HUANG, Y., LI, H., WANG, L., LIU, Y., PLUEMPITIWIRIYAWEJ, C., LEE, M., AND LAM, H. 2001. An internet-based negotiation server for e-commerce. *VLDB J. 10*, 72–90.

WINOTO, P., MCCALLA, G., AND VASSILEVA, J. 2002. An extended alternating-offers bargaining protocol for automated negotiation in multi-agent systems. In *Proceedings of the 10th International Conference on CoopIS* (Irvine, CA). Springer-Verlag, New York.

ZAHARIOUDAKIS, M., COCHRANE, R., LAPIS, G., PIRAHESH, H., AND URATA, M. 2000. Answering complex sql queries using automatic summary tables. In *Proceedings of the ACM SIGMOD'00 Conference* (Dallas, TX). ACM, New York, 105–116.

Received July 2004; revised August 2005 and January 2006; accepted January 2006