Replication-Aware Query Processing in Large-Scale Distributed Information Systems

Jie Xu
Department of Computer Science
University of Pittsburgh
xujie@cs.pitt.edu

Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

ABSTRACT

In this work, we address the problem of replica selection in distributed query processing over the Web, in the presence of user preferences for Quality of Service and Quality of Data. In particular, we propose RAQP, which stands for Replication-Aware Query Processing. RAQP uses an initial statically-optimized logical plan, and then selects the execution site for each operator and also selects which replica to use, thus converting the logical plan to an executable plan. Unlike prior work, we do not perform an exhaustive search for the second phase, which allows RAQP to scale significantly better. Extensive experiments show that our scheme can provide improvements in both query response time and overall quality of QoS and QoD as compared to random site allocation with iterative improvement.

1. INTRODUCTION

The Web has become the de-facto user interface and interconnection platform of modern life. Almost all collaborative, data-intensive applications are built for the Web or face obscurity. Many data-intensive applications are fueled by data from the physical world, thanks to the proliferation of (wireless) sensor technologies which are giving an unprecedented level of access and interaction with the real world.

In our Secure-CITI project (http://www.cs.pitt.edu/s-citi/), we envision a Web-based platform to be used to coordinate human response to disaster management. There is a pre-disaster component where different types of sensors are deployed in a networked fashion and are used to detect disasters (e.g., gas and water usage suddenly increase dramatically which could indicate a landslide in that area). There is also a critical component during the emergency, where in addition to sensor information, the same system is expected to be used to provide additional information (e.g., by providing real-time information about the capacity of area hospitals) and to coordinate human response (e.g., by identifying what is needed to perform a particular task and dynamically forming teams with the appropriate expertise to respond to it [2]). In such a scenario, many heterogeneous systems are glued together, facilitating the discovery and flow of critical information as a response to user requests.

To improve reliability, expedite data discovery, and increase performance, replication is expected to play a major role in large-scale distributed information systems, like the one we are exploring for Secure-CITI. By replicating information across multiple sites, crucial information can be accessed even in cases of disconnection or failure, which is common during disaster response (and most environments that are exposed to nature). Replication also allows for looser synchronization across multiple sites, which is necessary if the system spans different administrative or jurisdictional domains, which is typical in disaster management. In addition to data availability, replication allows for easier discovery of information, especially when catalogs are not present or not well maintained (i.e., the equivalent of unstructured overlay networks). Finally, replication is expected to drastically improve the overall performance of the system by reducing communication latency when requests are served locally or from close-by nodes. As such, we expect the most “valuable” data to be highly replicated across the entire system.

Although replication increases data availability and improves performance (i.e., Quality of Service, or QoS), it may have a detrimental effect to the Quality of the Data (QoD) that are being returned to the users. Getting results fast is crucial of course, but usually a limit to the degree of “staleness” is needed to make the results useful. Approaches for measuring QoD are traditionally grouped into three categories: time-based (where the time of last update is used), divergence-based (where the difference in value is used), and lag-based (where the number of unapplied updates is used) [9]. We concentrate on time-based measures of QoD, because we believe them to be the most general and the best fit for our case.

In this paper, we advocate going beyond simply measuring QoS and QoD. We introduce Quality Contracts (QC) as a novel way of specifying user preferences (with respect to QoS and QoD) and evaluating the system’s adherence to them. The QC framework utilizes a market-based mechanism, which has been used in the past to solve resource allocation problems in distributed systems [11]. As such, it provides a natural and integrated way to guide the system towards efficient decisions that increase the overall user satisfaction. The QC framework also enables users to describe the relative importance of different queries and also the relative importance of the different quality metrics (e.g., preference for fast answers that are slightly stale). This results in "socially" optimal solutions for the entire system.

Using the QC framework, we propose a Replication-Aware Query Processing (RAQP) scheme that optimizes query execution plans for distributed queries with Quality Contracts, in the presence of multiple replicas for each data source. Our scheme follows the...
classic two-step query optimization [12, 7, 8]: we start from a statically-optimized logical execution plan and then apply a greedy algorithm to select an execution site for each operator and also which replica to use. The overall optimization goal is expressed in terms of “profit” under the QC framework (i.e., the approach balances the trade-off between QoS and QoD), and as a special case, in terms of the traditional response time metric.

We provide the assumed system architecture and the QC framework in the next section. Section 3 contains the details of our RAQP scheme. We present extensive experimental results in Section 4. Section 5 describes related work. We conclude in Section 6.

2. SYSTEM OVERVIEW

2.1 System Architecture

We envision a large-scale distributed information system, where heterogeneity in all aspects is the norm. Such a system is expected to bring together (1) a myriad of receptors, that are sensing the environment (e.g., RFID readers or sensors), contributing a tsunami of information, (2) a high number of core nodes that are providing a stable communications, storage, and query processing substrate, and, a plethora of end-user access devices (e.g., mobile PDAs or light-weight desktop machines) that are enabling their users to collaborate, contribute data and knowledge and cooperatively work towards a common goal (e.g., disaster response).

We assume that each data item is "owned" by a specific node, but there is rampant replication in the system for availability and performance reasons. To facilitate data discovery, we build routing indexes[4] to direct the queries towards the nodes that are expected to hold relevant data. Each node maintains a local index, summarizing its local content. Core nodes maintain second-level indexes, that summarize index information from all nodes that connect to them, akin to a hybrid, unstructured P2P overlay network (e.g., Gnutella2). Core nodes can also exchange information among them, building merged indexes to summarize information on other reachable core nodes within a predefined horizon.

Query processing in our system is performed at core nodes. An end-user will send out a query message request along with a Quality Contract (QC) and a Time-To-Live (TTL) that controls the maximum number of hops the query could be routed in the network. After receiving a query message, a core node first checks its local index to answer the query, and then checks its merged index (in case a neighbor core node can answer the query). If there is a match (while abiding with the specified QC) an acknowledgment is sent back to the originator node which builds a query plan along with all the possible options for replicas. If there is no match, the query message is propagated further, until the TTL is reached.

2.2 Quality Contracts

In this work, we introduce Quality Contracts (QC) as a novel way of specifying user preferences (with respect to QoS and QoD) and evaluating the system’s adherence to them. The QC framework utilizes a market-based mechanism, which has been used in the past to solve resource allocation problems in distributed systems [11]. In our framework, users are allocated virtual money, which they “spend” to execute their queries according to their specifications for QoS (i.e., how fast they should get their results) and QoD (i.e., how fresh the results should be). Query servers (core nodes) execute queries and get virtual money in return for their services. The QC framework is general enough to also allow for refunds: servers pay virtual money back to the users if their queries were answered in an unsatisfying manner. In order to execute a query, both the user and the server must agree on a Quality Contract (QC). The QC essentially specifies how much money the user is willing to pay to get their queries executed (according to their specifications for QoS and QoD). The amount of money paid to the server depends on how well the query is fulfilled, based on the user’s preferences.

We model Quality Contracts as a collection of graphs. Each graph represents a QoS/QoD requirement from the user. The Y-axis corresponds to an attribute that the users want to use in order to measure the quality of the results, for example, response time. The Y-axis corresponds to the virtual money the user is willing to pay the server in order to execute his/her query. The QC graph is simply a function that maps quality (i.e., the user’s specification) to virtual money (i.e., the server’s reward). More than one graphs can be combined in a single QC, allowing for a user to consider multiple dimensions of quality of results, for example both QoS and QoD. The benefit of such approach is that users can easily specify the relative importance of the different quality metrics. For example, 75% of the “budget” of a given query can be allocated to QoS with the remaining 25% given to QoD; in this scenario, the system will give more attention to QoS instead of QoD (which will also be considered though). A server is expected to get “paid” equally to the sum of all the virtual money from the different QC graphs.

In this work, we use linear functions for QCs; this can easily be extended to any monotonically decreasing function. Each QC is composed of two quality metrics: response time (QoS) and staleness degree, measured as time elapsed since the last update (QoD). An example of two such QC graphs is given in Figure 1a and 1b. In this example, the user has allocated $75 for optimal QoS, and $25 for optimal QoD.

2.3 Query Execution Plan

As in previous work [5], we assume that statistics regarding the cardinalities of the relations and the selectivities of the operators are available to the optimizer. We also assume that we can estimate the transmission cost between nodes and that it is relatively stable, so that the query optimizer can approximate the transmission cost incurred in transferring data. To estimate the latency between two arbitrary Internet end hosts, we utilize approaches from the networking community such as IDMaps[6].

In this work, we extend traditional query execution plans with a transmission operator which will allow us to incorporate transmission costs in the query optimization process in a natural and integrated way. Transmission operators simulating transmission cost are simply incorporated between each pair of processing operators in the query execution plan. In this paper, we considered Select-Project-Join operators, however the framework and algorithms can be easily extended to handle non-relational operators, e.g., for querying XML documents (XQuery).
We present a QEP example in Figure 2. Processing operators and transmission operators are depicted as circles. Candidate sites (with replicas) are depicted as boxes. To generate an executable QEP, we need to select one site for each processing operator.

3. RAQP ALGORITHM

Our Replication-Aware Query Processing (RAQP) scheme follows the traditional two-step query optimization scheme[8] which has been extensively used in distributed database systems (e.g., XPRS[7] and Mariposa[12]). In the first phase, RAQP runs as if in a centralized system, generating the best query plan from a static point of view. In the second phase, RAQP dynamically chooses a replica for each data item and the execution site for each query operator, thus creating the final execution plan.

3.1 First Phase

In the first phase, we adopt the classic dynamic programming algorithm which has been proposed in system R[10]. However, we have lifted the constraint of left-deep trees from the system R optimizer and we are also looking at bushy trees, which is more appropriate in distributed systems, since they increase the parallelism.

3.2 Second Phase

In the second phase, we have two tasks to perform: replica selection (at the leaf nodes) and execution site selection (for the processing operators). We take the static query execution tree created from static planning phase as input and generate the physical execution plan as output.

Most previous work, assumed either that each data item has only one copy in the system, or that one replica has been selected prior to query optimization[8]. In most cases, the Replica Selection Problem (RSP) has been neglected. We expect high degree of replication in the types of systems that we are interested in (e.g., for disaster management), and as such, RSP is important. We define RSP formally as follows:

**Definition 1.** Assume a set of data items, \( D = \{1, 2, \ldots, m\} \), each of which has a set of replicas, \( S = \{R^1_{i}, R^2_{i}, \ldots, R^n_{i}\} \), where \( i \in D \). Each replica \( R^i_{i} \) is a tuple \((p^i_{i}, s^i_{i}, q^i_{i})\) where \( p^i_{i} \) is a price, \( s^i_{i} \) is the site replica \( R^i_{i} \) resides and \( q^i_{i} \) is the QoS of the replica. The Replica Selection Problem (RSP) attempts to label each replica as winning or losing, so as to maximize the processing node’s revenue under the constraint that exactly one replica of each data item needs to be selected to form the best allocation.

\[
\text{Max} \{B(alloc) - \sum_{s \in S} p^i_{s}X_s\}
\]

\[
s.t. \quad \forall s \in S, \quad X_s \in \{0, 1\} \quad \text{and} \quad \forall i \in D, \quad \sum_{s \in D} X_s = 1
\]

where \( B(alloc) \) is the sum of “profit” the processing node gets for fulfilling the quality contract.

Regarding site selection, previous work simply used exhaustive search to find the optimal allocation, since only one plan (which is selected by the first phase) will be explored. However, when we combine replica selection with site selection, exhaustive search is prohibitive for this NP-hard problem.

In our algorithm, we treat RSP and site selection in an integrated way. Our goal is to choose one site for each node in QEP, so that the final allocation is the best one in terms of either simply minimizing response time or maximizing the processing node’s profit which is measured by total benefit from QCs minus the money “spent” on acquiring data. Specifically, we take query response time as QoS and aggregated staleness for QoD. We aggregate staleness by selecting either the highest staleness over the top data items or the average staleness of the top-k data items. Currently we assume that all replicas of a given data item have the same price and the same size. It is outside the scope of this work to determine how to effectively select such prices.

3.2.1 Initial Query Allocation

Given a statically optimized query plan, we first determine the allocation ordering. At each processing node, we set a flag to indicate our estimation of which subtree is more likely to be the bottleneck.

Next, we will allocate that subtree first. Formally, we traverse the tree in post order, for each arc \( i \rightarrow j \), assign weight as \( \alpha \) times the transmission cost between \( i \) and \( j \) plus \((1 - \alpha)\) times the local processing cost at node \( i \). For each intermediate node, we calculate the aggregate working load from each subtree and mark \( M(i) \) to indicate the most expensive one. If it comes from left, \( M(i) = 0 \), otherwise \( M(i) = 1 \). This will be our optimization direction for future use. \( \alpha \in (0, 1) \) is a dynamic tuning factor to balance the transmission and processing cost. We use \( S \) in our experiment (i.e., giving equal importance). We provide the pseudo code for this MARK-OPT-DIR algorithm in Figure 3.
tri-alloc()  
1  j = L(child(i));  
2  k = R(child(i));  
3  //BW(i,j): bandwidth between i and j;  
4  α = 1/avg(C(i)), i ∈ S_j ∪ S_k  
5  β = 1/avg(BW(i,j)), i, j ∈ S_j ∪ S_k and i ≠ j  
6  Tr = β × max(Tr(i), Tr(k), Tr(C(i), C(k)))  
7  Tp = α × min(Lp(i), Lp(k))  
8  switch  
9    case Tr/Tp ≥ θ:  //Bandwidth – Bound  
10      if S_j ∩ S_k ≠ ∅  
11        then s = MC(S_j ∪ S_k)  
12          L(i) = L(j) = L(k) = s  
13        else BW(m,n) = max(BW(x,y))  
14          x ∈ S_j, y ∈ S_k and x ≠ y  
15          L(j) = m; L(k) = n  
16        if Tr(j,i) > Tr(k,i)  
17          then L(i) = m  
18        else L(i) = n  
19    case Tr/Tp < θ:  //CPU – Bound  
20      if LP(j) ≥ LP(k)  
21        then C(p) = max(C(s)), s ∈ S_j; L(j) = p  
22          C(q) = max(C(s)), s ∈ S_k – p; L(k) = q  
23        else C(p) = max(C(s)), s ∈ S_j; L(j) = p  
24          C(q) = max(C(s)), s ∈ S_k – q; L(k) = q  
25        C(r) = max(C(s)), s ∈ {S_j∪k – p – q}; L(i) = r  
26    return

Figure 4: Algorithm TRI-ALLOC

4. EXPERIMENTAL STUDY

We evaluated our proposed replication-aware query processing algorithm experimentally by performing an extensive simulation study using the following algorithms:

- **Exhaustive Search (ES):** Explore the whole search space exhaustively, thus guaranteeing to find the optimal allocation.
- **RAQP-G:** Greedy replication-aware initial allocation plus iterative improvement.
- **RAQP-L:** Bottleneck breakdown, local exhaustive search plus iterative improvement.
- **Rand(k):** Random initial allocation plus k steps of iterative improvement. In each step, the bottleneck node is identified and a random replacement is selected, if our optimization goal is improved. We use it as a "sample" of the search space.

We implemented an initial prototype of our distributed system, as described in section 2. The system parameters used in our experiments are reported in Table 1.

<table>
<thead>
<tr>
<th>Simulation Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Node Number</td>
<td>100</td>
</tr>
<tr>
<td>Edge Node Number</td>
<td>100</td>
</tr>
<tr>
<td>Unique Data Source</td>
<td>1000</td>
</tr>
<tr>
<td>Unique Data Number Per Data Source</td>
<td>U(10, 100)</td>
</tr>
<tr>
<td>Data Size</td>
<td>U(20, 200Mbps)</td>
</tr>
<tr>
<td># of Replicas Per Data Source</td>
<td>U(10, 30)</td>
</tr>
<tr>
<td>Bandwidth between each pair of Nodes</td>
<td>U(1, 50Mbps)</td>
</tr>
</tbody>
</table>

Table 1: Default System Parameters in Experiments
4.1 Optimizing for Response Time

In the first set of experiments, our optimization goal is to minimize the response time. We evaluate the different optimization algorithms under various circumstances. To avoid bias in the results, we repeated each experiment 5 times with different random seeds, and report the average values.

4.1.1 Effect of Replication Degree

We artificially varied the number of replicas per data item in the system. We optimize queries with 6 joining relations. The quality of the resulting plans is reported in Figure 5. Clearly, our RAQP algorithms greatly outperformed Rand(5), and RAQP-L was a bit better than RAQP-G. As expected, ES always finds the optimal plan.

An obvious observation is that the quality of the resulting plans is improved as the number of replicas is increased. There are two reasons. First, as the number of replicas increases, we get more nearby candidate sites (which improves response time). Second, more replicas mean more candidate sites for execution, which increases the search space (and improves the overall response time).

4.1.2 Optimization Overhead

Exhaustive search can always find the optimal plan, however, its running time is prohibitive in large-scale systems. We fixed the number of replicas to 20 and looked at the running time of different algorithms versus the quality of the resulting plans. We report our findings in Table 2(a). Clearly, ES took around 2 days to find the optimal plan, while our RAQP-G algorithm took around 70 ms.

We also report the optimization time for queries with 3 and 1 joins in Table 2(b) and 2(c). We found similar trends in all three experiments. We did not increase the number of joins beyond 6 as we would not have been able to compare with the exhaustive algorithm.

4.1.3 Effect of Network Size

In the second set of experiments, we varied the network size to observe the changes in optimization quality. In order to show the trend clearly, we chose 100 to 400 as the network size. The results are reported in Figure 6. A jump, from 100 to 200 makes small difference in the relative quality change. However, when the network size increased to 300, 400, the random algorithm showed much worse performance compared to others. The reason is when the system enlarges, our search space becomes sparser. Not surprisingly, the blind random search performed much worse than our informed heuristic search. For the above sets of experiments, we also confirmed that our algorithm scaled very well in both running time and the optimization quality. RAQP is also relatively stable under various network size and data loads.

4.2 Optimizing for Profit

In this set of experiments we tune our algorithms to optimize for the overall QC profit instead of simply response time.

One of the important features Quality Contracts hold is that users can easily specify the relative importance of each component of the overall quality by allocating the query budget accordingly. In order to observe the algorithm performance under different environments, we classify the users’ quality requirements into 6 classes. We have three values for QoS and QoD: high (75), low (25), same (50) and two types of slope for the QC function: small and large, which produce 6 separate classes. Each data item had 20 replicas. We report our results in Figure 7.

Since our allocation initialization algorithm was aimed at response time improvement, QoS got more improvement than QoD in all the cases. Especially when QoS was assigned higher budget, the effect on both QoS and total profit were obvious. When QoD was assigned higher budget, the relative improvement of QoD also increased compared to the lower budget case.

Our results clearly confirmed the functionality of Quality Contracts and our RAQP algorithm. Assigning higher "budget" to a quality dimension ends in that dimension getting better performance by our optimization algorithm. The larger the budget difference the larger the difference in the resulting quality. This behavior is unique to our algorithm and is an important feature to have when both QoS and QoD are of concern to users.

5. RELATED WORK

Mariposa[12] is the first distributed DBMS to use economic schemes as the underlying paradigm. Queries are submitted to the Mariposa system with a bid curve on delay; a broker sends out requests,
works fairly well for both optimization goals, response time and total profit from multi-dimensional quality requirements. In this paper, we focused on SPJ queries, however, we believe the proposed framework and algorithms can be easily extended to cover XML data and XQuery.

Another system close to ours is Borealis[1], a distributed stream processing engine, which inherits stream processing functionality from Aurora[3]. Aurora adopted a utility-function based QoS model on the processing delay of output tuples. Borealis also uses multi-dimensional quality metrics which could include response time, quality of data and etc. The difference from our work (QC) is that Borealis simply measures the overall quality by calculating an aggregated value from a global weight function, and the weight for each quality dimension is fixed. In our work, each user can allocate these weights and differentiate among queries and quality metrics.

There is a lot of work on distributed query optimization, but to the best of our knowledge, we are the first to address the problem of replica selection under quality guarantees.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced Quality Contracts (QC) as a unifying framework to enable users to specify their QoS and QoD requirements and their relative importance. Additionally, we proposed a replica-aware query processing scheme and demonstrated that it works fairly well for both optimization goals, response time and total profit from multi-dimensional quality requirements. In this paper, we focused on SPJ queries, however, we believe the proposed framework and algorithms can be easily extended to cover XML data and XQuery.

7. REFERENCES


