Wee Siong Ng Beng Chin Ooi Yan Feng Shu Kian-Lee Tan Wee Hyong Tok
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 17543
email: {ngws, ooibc, shuyanfe, tokwy, tankl}@comp.nus.edu.sg

Technical Report NUS-CS01-03

January, 2003

Abstract

In this paper, we propose a distributed continuous query (CQ) processing system based on Peer to Peer architecture (P2P) technology, where peers collaborate in terms of processing and providing data. In particular, we propose that data streams be reallocated and query sharing be exploited at the peer-to-peer level. Our proposed approach supports a novel class of queries, which we call *pervasive CQ*. A pervasive CQ issued by peer A is evaluated by peer B and stored there to be retrieved by peer A at a later time. This class of continuous queries is particularly beneficial in the heterogeneous environment in which peers operate. We conduct an extensive performance study to evaluate the proposed strategies, and our results show the effectiveness of the proposed schemes.

Contact Author:
Wee Siong Ng
Department of Computer Science
National University of Singapore
Science Drive 2, Singapore 17543

Office: (65) 6874-4774
Fax: (65) 6779-4580

http://www.comp.nus.edu.sg/∼ngws
Email: **ngws@comp.nus.edu.sg**

Efficient Distributed Continuous Query Processing using Peers

Wee Siong Ng Beng Chin Ooi Yan Feng Shu Kian-Lee Tan Wee Hyong Tok
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 17543
email: {ngws, ooibc, shuyanfe, tokwy, tankl}@comp.nus.edu.sg

Abstract

In this paper, we propose a distributed continuous query (CQ) processing system based on Peer to Peer architecture (P2P) technology, where peers collaborate in terms of processing and providing data. In particular, we propose that data streams be reallocated and query sharing be exploited at the peer-to-peer level. Our proposed approach supports a novel class of queries, which we call *pervasive CQ*. A pervasive CQ issued by peer A is evaluated by peer B and stored there to be retrieved by peer A at a later time. This class of continuous queries is particularly beneficial in the heterogeneous environment in which peers operate. We conduct an extensive performance study to evaluate the proposed strategies, and our results show the effectiveness of the proposed schemes.

1 Introduction

Continuous queries (CQ) are queries that are executed for a potentially long period of time, and are used in the monitoring of data semantics in the underlying data streams to trigger user-defined actions. Continuous queries transform a passive networked structure into an active environment, and are particularly useful in distributed environments where huge volumes of information are updated frequently and remotely. For example, users may be interested in monitoring the trading volume or price of a particular stock over a period of time. They could then express their request in a continuous query as follows:

Notify me whenever the trading volume increases 2% OR Oracle stock increases by more than 5% over the next three months.

Figure 1: Example CQ query.

Peer-to-Peer architecture (P2P) is emerging as a new paradigm for information sharing. In a P2P distributed system, a large number of nodes (e.g., PCs connected to the Internet) can potentially be pooled together to share their resources, information and services. Unlike traditional client-server architecture, P2P allows peers to publish information and share data with other peers without going through any intermediate machines. As a result, the information that is accessible to individuals grows at a faster and larger scale.

In the literature, much of the existing work that seeks to support a large number of potential CQ requests, focuses on efficiently handling the processing of a large number of continuous queries. Most of that effort centers on finding an optimal plan for similar queries or subsuming a new incoming query into an existing queries group [1]. These existing techniques, however, are not expected to perform well in a highly distributed environment for several reasons. First, these techniques were designed mainly based on a centralized client-server architecture. Queries are routed and registered to a central continuous query system (CQS). Thus, much of the existing work focuses on supporting as many queries as possible against external data sources. However, it is clear that there is a limit to the number of queries that can be handled by a single server, no matter how efficient the CQS may be. Second, most of these techniques focus on the data stream consumer (i.e. the system processing the continuous queries), and neglect that the data stream providers do have a significant impact on the responsiveness of the CQS. A popular data stream provider may be easily overwhelmed by requests and consequentially delay the response of a CQS. Third, there exists no computational sharing among CQS. Each of the server is autonomous and performs optimization for queries that are registered to them. Queries are merged into existing local groups and no knowledge is shared among CQS [1]. Thus, much of the work performed by individual CQS is duplicated. In addition, resources at some CQS are under-utilized. For example, a large number of CQS may be accessing the same data source, thus overloading the data sources providers. Furthermore, existing CQS is designed to deliver results as they are computed. However, there are many situations in which continuous delivery of results may be infeasible and impractical. For example, Data Recharging [2] describe a process through which personal mobile devices such as PDAs connect to the network and refresh its contents periodically. Clearly, with the limited connection time to the data source, continuous delivery is inefficient.

In this paper, we propose a distributed continuous query processing system based on P2P technology. Peers collaborate in terms of processing and providing data. In particular, we propose that data streams be reallocated and query sharing be exploited at the peer-to-peer level. The proposed approach supports a novel class of queries, which we call *pervasive CQ*. A pervasive CQ issued by peer A is evaluated by peer B and stored there to be retrieved by peer A at a later time. This class of continuous queries are particularly beneficial in the heterogeneous environment in which peers operate. In the existing P2P contexts, all devices (i.e. peers) may differ both in hardware and software configurations, as well as computational capabilities [3, 4, 11, 17]. For example, a mobile device such as a Personal Digital Assistant (PDA) has limited computational power and memory compared to a desktop machine. Clearly, a PDA has limited functionality compared to a desktop machine. *pervasive CQ* leverages on this difference, relying on "stronger" peers to compensate for the physical limitations of "weaker" peers. It also allows peers to disconnect and rejoin the network without any restriction, and without any loss to the users in terms of access to requested information.

The rest of this paper is organized as follows: in the next section, we discuss how we can make use of a large network of peers for continuous query processing. In addition, we discuss issues that are prevalent when applying P2P to the domain of continuous query processing, and we present solutions. In Section 3 we present the CQ-Buddy, a distributed CQS. Section 4 reports the findings of an extensive experimental study to benchmark the performance of CQ-Buddy. We review related work in Section 5, and finally, we conclude in Section 6.

2 Towards P2P Continuous Query Processing

As noted in the introduction, practically all existing CQ systems are designed and implemented based on a client-server architecture, and are not expected to scale well in a distributed environment for these reasons: (i) the data provider may become a bottleneck, (ii) there may be duplicate processing among nodes, and (iii) the operating environment is heterogeneous. In this section, we discuss how distributed CQS can be realized using P2P technology. For this purpose, we shall refer to a node in the distributed network as a *peer*. We shall briefly discuss the basic approaches here, and leave the detailed discussion and strategies to subsequent sections. We address the problem in three different aspects. First, we study how data streams reallocation can be employed in a CQS to reap substantial performance improvement. Second, we consider similar queries sharing and queries grouping at a peer-to-peer level. Finally, we introduce the notion of pervasive continuous queries and show how peers can benefit from the pervasiveness.

2.1 Data streams reallocation

When a large number of peers access the same data source, the data provider becomes a bottleneck. There are two strategies that can be adopted to alleviate this problem. We refer to these strategies as *REDIRECT* and *ISO-PEER*.

In the *REDIRECT* strategy, a peer re-directs a new request to other peers that can provide the same data requested. The issues here are three-fold. First, a criterion has to be determined on when redirection should be performed. Second, we need to know which peer to redirect the request to. Finally, there is a need to manage the information, as it may not be easy to locate peers that can provide the same data.

In the *ISO-PEER* strategy, a peer delegates several other peers (amongst the large number of peers accessing it) as intermediate peers, and channels requests of access to these intermediate peers. We refer to these intermediate peers as *iso-peers*, as each of these peers is fetching the data on behalf of other peers and is in essence providing data as the original data provider. The *ISO-PEER* strategy shares the first two issues facing the *REDIRECT* strategy: the need for a criterion on when redirection should be performed, and the need to know which peer to send the request to. Similarly in both strategies, the data-providing peer reduces the number of peers accessing it simultaneously. In this way, both strategies reduce the workload of the data-providing peer and thus improve its performance. The ISO-PEER strategy, however faces one issue less - it does not require a peer to manage information about data providers. For this reason we will focus on the ISO-PEER strategy in this paper. Let us consider the following example:

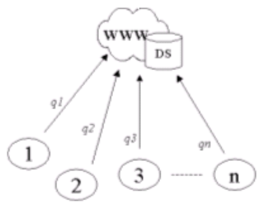
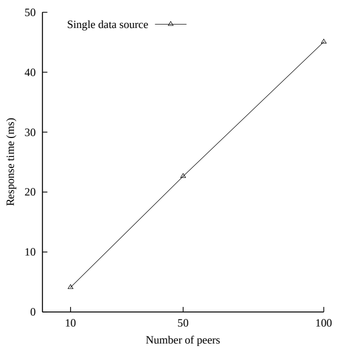


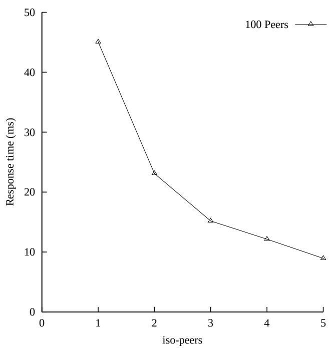
Figure 2: Multiple peers accessing a popular data source.

In Figure 2, we have n peers each issuing continuous queries to a popular peer. The peer quickly becomes a bottleneck, since it has to handle multiple query requests from multiple peers and send individual responses to each of them. We conduct a preliminary study to validate this example. In this simple experiment, we created a total of 100 peers (varies from 10 to 100). Each peer submits 50 queries on runtime to CQS. In the first set of experiment, we use a single stream provider and record the average response time of peers (see Figure 3(a)).

In the second set of experiments, we use the data stream reallocation strategy in which a primary data source delegates stream to n number of intermediary peers, i.e., iso-peers. Queries are submitted to these peers in a random manner. Figure 3(b) shows that the response time improves significantly. This is expected as each peer is doing less computation. Thus, we can clearly see the benefits of the data stream reallocation strategy.



(a) Response time vs. peers.



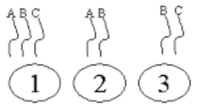
(b) Response time vs. iso-peers.

Figure 3: Benefits of stream reallocation.

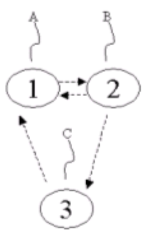
2.2 Resource sharing strategies

P2P technology facilitates the sharing of data and computing resources. Thus, we believe it can enhance the reliability and performance of a distributed CQS. Figure 4(a) illustrates a scenario where several "selfish" peers do not share the processing of continuous queries with their neighbors, and choose to process them by themselves.

On the other hand, in Figure 4(b), each peer does not handle the entire CQ processing of its own query; instead, it shares the processing workload with other peers in its neighborhood. The advantage that "sharing" peers have over selfish peers is obvious: each peer helps one another by processing the data for others. In the figure, we have Peer 1 processing data stream A that is also required by Peer 2. Peer 1 will only forward the data to Peer 2 when the data passes the filtering criteria defined by Peer 2. Similarly, Peer 2, which is processing stream B, will only forward data to Peer 1 and 3 if the data passes the filtering criteria defined by these peers.



(a) Selfish Peers.



(b) Sharing Peers.

Figure 4: Peers' Relation.

We shall defer the discussion on a formal definition of similar queries and how each peer can detect similar queries to a later section.

2.3 Heterogeneous operating environment of Peers

In a heterogeneous operating environment, peers can reside on a limited resource device such as a PDA, as well as on a desktop or notebook. The basic idea is to allow peers that are "weaker" than its neighboring peers to ask buddy peers for "help" to process either the entire query or a fragment of the original continuous query.

2.4 Frequent connection/disconnection of peers

Before leaving this section, let us look at an example that motiviates the concept of *pervasive CQ*. Consider a traveler T visiting Country X, who wish to stay apprised of information such as the trading volumes of financial markets with a volume greater than certain threshold. T requests the peer software running on his PDA to perform the following simple continuous query:

```
SELECT stock.symbol, stock.volume_traded
FROM nyse.stream
WHERE stock.symbol = 'ORA' or 'SUN' AND stock.volume_traded > 1000
AT every interval of 10 minutes
STORE only top 5 volumes
```

When T boards the plane, his PDA is disconnected from the network of peers. However, before disconnecting, his peer software asks for "help" to perform the query amongst the buddy peers. When he arrives in Country X, he powers up his PDA and immediately, the buddy peers provide him (rather his PDA) with the information he requested. Two interesting characteristics observe in this query which its support are absent from the existing CQS implementation. First, the results need to be stored and must be downloaded to his PDA when it is connected to the network. Second, even when online, he might only wish to see the recent movement financial activities, e.g., STORE only top 5 volumes, rather than interrupted by every update.

We refer to this class of continuous queries that are processed by a peer on behalf of another peer, and retrieved at a later time period as *pervasive continuous queries.* In a later section, we show how such queries can be supported.

3 CQ-Buddy: A Distributed CQS Using Peer Technology

In this section, we shall present CQ-Buddy, a distributed CQS that employs P2P technology extensively. We shall first look at the CQ-Buddy network and the architecture of a CQ-Buddy node. Then, we shall present the strategies for sharing computation and queries among peers.

3.1 CQ-Buddy Network

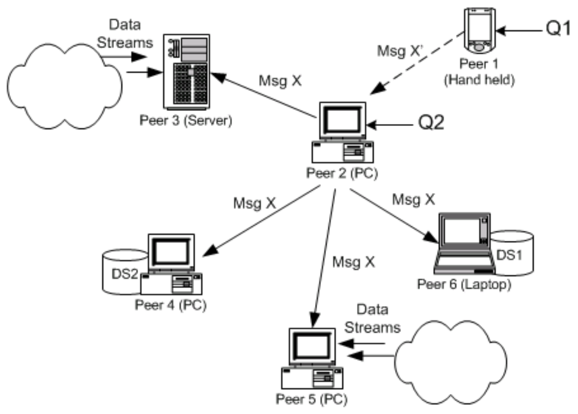


Figure 5: Overview of CQ-Buddy Network.

CQ-Buddy is a P2P-enabled distributed CQS. The network consists of two kinds of peers. First, peers with CQ processing capabilities but which do not provide data streams to other peers. Second, peers with CQ processing capabilities and which provide data streams to other peers. We refer to the former as CQC (CQ consumer, e.g., Peer 1 and Peer 2 in Figure 5) and the latter as CQD (CQ data stream provider, e.g., Peer 3 to Peer 6 in Figure 5).

All incoming queries that are submitted by the user are first optimized by the CQC internally, e.g., forming them into grouped similar queries. Following that, the CQC submits the queries or grouped queries to the CQD, to see whether there are other peers (i.e. CQ-Buddies) who can help in the processing. This hypothetical model is practical especially in a P2P environment, where some peers are more reliable and stable than the others, e.g., workstations as compared to PDAs, and dedicated network lines as compared to modem dial-ups. In such pairs, the "weaker" peer acts as the CQC instead of the CQD.

In our model, a continuous query server (CQS) consists of two components: optimization (i.e. grouping of similar queries), and evaluation. The CQC essentially performs the role of the first part, while the CQD can be viewed as a CQS. The difference between the two is that the CQD is more stable, and does not typically disconnect. Each CQC can act as an intermediate node for other peers, and also be consuming data provided by the CQD.

For illustration, Figure 5 shows a CQ-Buddy network with several heterogeneous peers, including a handheld device (Peer 1), laptop (Peer 6), PCs and a server-type peer. Assume Peer 1 and Peer 2 are CQC, Peer 3 and Peer 5 are the original CQS (we assume there exist some existing and permanent CQS) and the rest are CQD. An incoming query Q2 = *select * from nyse.stream where Stock.symbol ='MSN' or Stock.symbol = 'ORA'* is submitted to Peer 2. Since Peer 2 is a CQC, it is unable to process the incoming query on its own. Hence, a Msg X = *Who can handle query select * from nyse.stream where Stock.symbol = X?* will be sent out to the peers' network. Note that the objective is to locate peers which currently handle *similar process* (i.e., monitoring data source nyse.stream with projection attributes Stock.symbol), so no exact match of projection attributes is necessary. When a CQD receives a request, it may either handle the query if it has the similar queries running in its local process pool, or drop the message otherwise. Msg X keeps on propagating to neighboring peers and the live time is controlled by TTL (Time-to-Live). TTL indicates the maximum number of hops the message can be passed on before it expires, and this is used to avoid flooding the network. There is also a mechanism for breaking the message loops: each peer keeps a queue of the recent messages and rejects the ones that have been processed before. CQDs which are able to handle the query (i.e. able to merge the incoming query into the existing process group) will send an acknowledgement directly to Peer 2 with its identity, BPID [1]. Peer 2 keeps the BPIDs, which may be used for further reference, e.g., to remove the query.

Peer 2 has no advance knowledge of the number of CQDs that will respond. Instead, it relies on a predefined threshold (e.g., stop when 2 CQDs return results or when timeout sets in). In the case of an empty result, the query will be sent to the original CQS, e.g., Peer 3 and Peer 5. A new process will be created in the process pool of Peer 3 and Peer 5 since there are no similar queries that are currently running. Note that although Peer 3 and Peer 5 can always process the incoming query (either merge it into the existing local process pool for similar queries, or create a new process to handle it), that option will only be taken last in order to avoid building up a single data source bottleneck.

Consider another query Q1 that is defined as Q1 = *select * from nyse.stream where Stock.symbol ='MSN' or Stock.symbol = 'ORA' STORE = 30 minutes.* It is similar to Q2 but with the additional parameter "STORE". This indicates that Q1 is a pervasive query and the peer which handles the query will help to store the result for the past 30 minutes. Peer 1 submits the pervasive query, Q1, to Peer 2 and disconnects after receiving an acknowledgment from Peer 2 (denoted by dash line).

[1] CQ-Buddy is built on top of BestPeer [12], BPID is a global identity used in BestPeer to uniquely identify different peers and their respective location in the dynamic network.

Peer 2 handles the query if it is a normal query in the manner as described previously. However, it has the additional task of helping Peer 1 to store the results and return the results to Peer 1 when Peer 1 reconnects. In the next section, we describe in detail the components of our architecture, and present how similar queries among peers are handled.

3.2 Architecture of a Peer Node

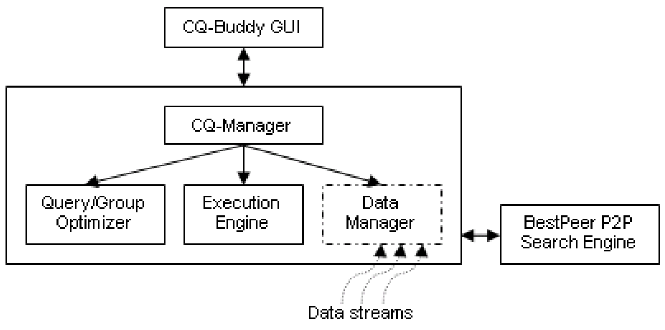


Figure 6: Architecture of a peer.

Figure 6 depicts the architecture of an autonomous peer in CQ-Buddy. CQ-Buddy is an extension of the BestPeer platform that provides low-level P2P facilities, e.g., communication, and search mechanism. The core of a peer in CQ-Buddy is the CQ-Manager that accepts user queries through a user interface and then invokes the underlying execution engine. Each query is optimized by the Query/Group optimizer, where it is integrated into a group of queries if it is similar to them. In the case of the CQC, an incoming query will first be optimized internally as described in the previous section. The queries or grouped queries will then be used as input for the P2P search engine to locate the CQD that can handle the queries. Note that the Data Manager module may not be operational in a CQC, since it simply consumes data provided by CQD or acts as an intermediate node for other peers. The data manager in a CQD monitors data sources in the local disk. The data source can be a flat file or any other data output from sensors. It is responsible for notifying the CQ-Manager of any modification of data. There are two possible actions when changes are detected. First, CQ-Manager invokes the execution engine to evaluate the installed continuous queries. Second, the CQ-Manager pushes the changed data to intermediate peers. In the following, we describe the mechanism for sharing similar queries among peers.

3.3 Sharing computation amongst similar queries

3.3.1 Similar queries among peers

In the literature on continuous query systems (CQS) [9, 1, 19], one of the most frequently tackled issues is the need to handle a large number of queries effectively and efficiently. Of the large number of continuous queries issued by users, many of the queries are similar, and by detecting these similarities, the queries can be processed collectively. Since the computation is shared, the resources allocated to process the queries can be significantly reduced. However, it must be noted that most of the existing work focuses on the sharing of computation in a single CQS.

Let us now consider a large network of peers, where each peer has continuous query-processing capabilities. There are more opportunities for the sharing of computation for similar queries amongst the peers. Let us first define similar queries as follows:

Definition 1 *Let us denote a selection predicate, $pred_j$, as $Col_j \circ_j X_j$ where $\circ_j$ denotes an operation in the set $\{\leq, \geq, \neq, =\}$, Col_j denotes a field name, X_j denotes a constant expression, and $1 \leq j \leq n$, where n is the number of selection predicates specified in a single query. Then, two selection predicates, $pred_1$ and $pred_2$ are similar if and only if*

1. $Col_1 = Col_2$ and $\circ_1 = \circ_2$

Definition 2 *A query Q_i, consists of a set of selection predicates S_i, a set of projection attributes P_i, and a set of data sources D_i. Let s_i denote the total number of selection predicates specified in S_i, d_i denote the number of data sources referenced in D_i, and p_i denote the number of projection attributes in P_i.*

Definition 2.1 *Given two queries, Q_1 and Q_2, the predicate similarity between Q_1 and Q_2 is defined as:*

PredSim(Q_1, Q_2) = s / max(s_1, s_2), where s is defined as the number of predicates in S_1 that are similar to the predicates in S_2

Definition 2.2 *Given two queries, Q_1 and Q_2, the similarity between the projections attributes in Q_1 and Q_2 is defined as:*

ProjSim(Q_1, Q_2) = p / max(p_1,p_2), where p is the number of projection attributes in P1 that are the same as the projection attributes in P_2.

Definition 2.3 *Given two queries, Q_1 and Q_2, the similarity between the data sources in Q_1 and Q_2 is defined as:*

DSSim(Q_1, Q_2) = d / max(d_1,d_2), where d is the number of data sources in D_1 that is the same as the data sources in D_2.

Example I: Given two queries Q_1 and Q_2 as follows:

*Q_1: select * from R where R.a = 5 and R.b = 3*

*Q_2: select * from R where R.a = 3 and R.c = 2 and R.b = 4*

PredSim(Q_1, Q_2) = 2 / 3

Example II: Given two queries Q_1 and Q_2 as follows:

*Q_1: select * from R where R.a = 5 and R.b = 3*

*Q_2: select * from R where R.a = 3 and R.b = 4*

PredSim(Q_1, Q_2) = 2 / 2 = 1

Definition 3 *A query Q_i, consists of a set of selection predicates S_i, a set of projection attributes P_i, and a set of data sources D_i. Given two queries, Q_1 and Q_2, the query similarity between Q_1 and Q_2 is defined as:*

$QuerySim(Q_1, Q_2) = (PredSim(Q_1, Q_2) + ProjSim(Q_1, Q_2) + DSSim(Q_1, Q_2))/3$
When QuerySim(Q_1, Q_2) = 1, the queries are similar to one another.
When QuerySim(Q_1, Q_2) = 0, the queries are not similar to one another.
When 0 < QuerySim(Q_1, Q_2) < 1, the queries are potentially similar to one another.

3.3.2 Query subsumption

When a newly introduced query is similar (by the above definition) to one of the queries in the existing pool of running queries, they can be processed collectively. Let us consider the following three queries. Suppose Query 1 is a newly introduced query, whereas Query 2 and Query 3 are queries that are in the existing pool of running queries. We can see that these three queries can be collectively processed, instead of processing them separately.

```
Query 1:
SELECT stock.symbol, stock.volume_traded FROM nyse.stream
```

```
WHERE stock.symbol = 'ABC' or stock.symbol = 'CDE'

Query 2:

SELECT stock.symbol, stock.traded_price FROM nyse.stream

WHERE stock.symbol = 'EFG'

Query 3:

SELECT stock.symbol, stock.high FROM nyse.stream

WHERE stock.symbol = 'CDE'
```

Each of the three query results can be re-written as a subset of the following query, Q: where X = {'ABC', 'CDE', 'EFG'}

```
Query Q:

SELECT stock.symbol, stock.volume_traded, stock.traded_price, stock.high

FROM nyse.stream

WHERE stock.symbol = X
```

Queries 1, 2 and 3 are potentially similar queries that share the same data source. In addition, a superset of projection and selection attributes (as shown in Query Q) can be defined to cover the projection and selection attributes needed by the three queries.

3.3.3 Strategies for processing similar queries

When a peer receives a new continuous query for processing, it first determines whether the continuous query is similar to any of the queries running in its existing pool. The similarity between a newly arrived continuous query and all the running queries is computed. If the newly arrived query is similar to one of the existing running queries, it will be added onto the existing query. If the newly arrived query is similar to none of the existing running queries, the peer can choose from two strategies.

In the first strategy, which we refer to as SELF-HELP, the peer initiates a new processing task to handle this new query itself. In this manner, the peer behaves exactly like a single CQS. In the second strategy, which we refer to as BUDDY-HELP, the peer asks its buddy peers for "help" in processing the query. The buddy peers then process the query on behalf of the peer, and provide the peer with the results of the continuous query. In Section 4, we perform an extensive study on the effectiveness of these two proposed strategies.

3.4 CQD selection policy

When a CQD is over-loaded, it will pass a list of delegated CQDs to the requester (CQC). From the list, the CQC makes a selection and submits queries to the selected CQD. It should be obvious that the selection process is crucial to the processing performance of a CQ. Each CQD may be different in terms of their processing power and resources, which would influence the performance of CQ query processing. Let assumes given a list of m CQD candidates, two naiive solutions can be employed. First *Random* policy, in which the probability for selecting any CQD is equal to $1/m$. Second, *Round Robin* policy, works on a rotating basis in that one CQD_i is selected and used to process queries, then moves to the back of the list; the next CQD_{1+i} is selected, then it moves to the end of the list; and so on, until CQD_m is selected. This works in a looping fashion. However, these policies have never taking into consideration of giving preference to those CQD with the least amount of congestion or workload.

We propose a CQD admission and selection algorithm, called *Adaptive-L*, based on a randomized resource allocation technique called lottery scheduling [20] and taking into consideration of current load and the processing power of a CQD ahead of the submission of CQ query. The pseudo code for the *Adaptive-L* is presented in Algorithm 1.

Adaptive-L receives a candidate list $\underline{\omega} = (O_{oid_1}, O_{oid_2}, ..., O_{oid_l})$ as its input. The CQD which offers to process the query is denoted as object O_{oid} in the list $\underline{\omega}$. For each O_{oid} in $\underline{\omega}$, a short ping query will be sent to it. The ping query response time τ will be captured (Ref:1). In order to compute the ticket volume $v(O)$, τ will be used for function *computeVolume* as in Ref:2, which is a normalization mapping of the response time into a internal scale. If O_{oid} exists in the local cache, the volume will be combined by getting the mean value of the new volume value and cache volume value. Finally, *generateToken* step of Ref:3 is a random function that generates a token in the range of total sum of ticket volume. This is used to determine which corresponding O_{oid} will be selected as the final candidate.

3.5 Pervasive continuous queries processing

When the client (scenario: using a PDA) issues a pervasive continuous query, he indicates how long the data will be stored. He may request to store the data for the past X minutes. The data stored by the buddy is constantly refreshed, and thus kept up-to-date. The motivation for introducing a new class of pervasive continuous query is to allow devices such as PDAs to be able to issue a query, go to sleep (to conserve energy), or disconnect (such as when the user is travelling from one geographical location to another). Pervasive continuous queries provide users with the convenience of switching on their device again after being disconnected for a period of time, and finding at that point in the time the data that they have requested for earlier.

Algorithm 1: Adaptive-L($\underline{\omega}$)

```
    Data   : a candidate list $\underline{\omega} = (O_{oid_1}, O_{oid_2}, ..., O_{oid_l})$ of response $CQDs$ whose are able to
             process the query.
    Result : A selected $O_{oid_x}$ object.
    begin
        $n\underline{\omega} \longleftarrow \emptyset$, $\tau \longleftarrow 0$
        for $i \longleftarrow O_{oid_i} \in \underline{\omega}$ do
1           $\tau = roundTrip(O_{oid_i})$
            Let $T(v, O)$ be a ticket with volume $v$ for an object $O$
            Let $v(O)$ be the ticket volume for the object $O$
            Let $vc(O)$ be the ticket volume for the object $O$ that might be found in local cache
2           $v(O_{oid_i}) = ticketVolume(\tau)$
            if $vc(O_{oid_i}) \neq nil$ then
                $newVolume = (v(O_{oid_i}) + vc(O_{oid_i}))/2$
            else
                $newVolume = v(O_{oid_i})$
            $n\underline{\omega}[i] \longleftarrow$ new $T(newVolume, O_{oid_i})$
        $n\underline{\omega} \longleftarrow$ sorted $n\underline{\omega}$ in ascending $T.v$ order
        $a\underline{\omega} \longleftarrow \emptyset$, $av \longleftarrow 0$
        foreach element $e$ of the $n\underline{\omega}$ do
            $av \longleftarrow av \cup e.v$
            $a\underline{\omega}[i]$ = new $T(av, e.O)$
3       $token = generateToken(a\underline{\omega}[last])$
        $next \longleftarrow 0$
        while $token \leq a\underline{\omega}[next].v$ do $next++$
        return $a\underline{\omega}[next].O$
    end
```

4 A Performance Study

4.1 Cost Model

Let us denote a CQC as P. In addition, $Freq(q)$ is the denotation of the average frequency in which the CQC submits queries to a CQD, D (which is itself, a peer too). Assume n queries (Q_1,...Q_n) and each of the query takes E processing time in order to fulfill the request, i.e., submitting and

Parameter	Value	Comments
TR_R	3.68891 KB/sec	Average transfer rate between remote peers (WAN)
TR_D	4675.945 KB/sec	Average transfer rate from the disk
AMT_R	1.2975 sec/mes	Average time per message between remote peers (WAN)
ICT_R	3.68 sec/con	Average time to initiate a remote connection (WAN)

Table 1: Parameters derived from the prototype

processing in CQD. Let t be a result tuples and $size(t)$ defines the tuple size. The network cost N for transferring t from peer D to peer P is:

$$N(t, D \to P) = Cn(D \to P) + \frac{size(t)}{Tr(D \to P)}, \tag{1}$$

where $Cn(D \to P)$ is the cost of establishing a connection between the two peers and $Tr(P \to D)$ is the transfer rate between D and P. Hence, the processing cost, $c(p)$, for peer p, is defined as:

$$c(p) = \sum_{i=1}^{i=n} \left[Freq(q_i) \times (E_i + N\,(t, D \to P_i))\right] \tag{2}$$

The $c(p)$ defines the traditional centralized approach for processing incoming data stream requests. Suppose D delegates a stream to a set of peers as intermediate peers. It then redirects requests to these intermediate peers instead. The cost of peer p will be defined as $c'(p)$. The number of requests will decrease when more requests are redirected to newly delegated nodes [2]. Now, by delegating the processing load to intermediate peers, the value of $m < n$, and hence $c'(p) < c(p)$.

$$c'(p) = \sum_{i=1}^{i=m} \left[Freq(q_i) \times (E_i + N\,(t, D \to P_i))\right] \tag{3}$$

Overall, the total cost of the CQC will be determined by C_{cq} where the costly CQD is always the bottleneck of the system (i.e., each peer is autonomous and work in parallel). Obviously, in an environment with a single CQD, $n = 1$, hence itself becomes the bottleneck.

$$C_{cq} = \max_{i=1\ldots n} \left[c'(p_i)\right] \tag{4}$$

4.2 Experiment Setup

CQ-Buddy is built based on the BestPeer [12] architecture. For a more realistic simulation, we employ real-life parameters (see Table 1, which were also used in [8]). These parameters are subsequently used by a simulator to evaluate the behavior of CQ-Buddy.

Recall Definition 3, the similarity of two queries $Q1$ and $Q2$ is defined by $QuerySim(Q_1, Q_2) \in \{0, 1\}$. In our experiments, we introduce a parameter, called degree of overlap, which is denoted

[2]since redirection is a simple process, we assume it incurs zero cost in our model

as $\alpha \in \{0, 1\}$. The parameter α is the probability value used to determine whether the incoming queries are similar with existing queries in the local queries pool. When $\alpha = 0$, there is no overlap between the incoming query and existing running queries, and all queries are different. When $\alpha = 1$, each incoming query is similar to one of the running queries.

4.2.1 Data sets

We run our experiments against two different data sets, R and S. Each relation consists of 10,000 tuples, and we assume every join query in our experiments is a one-to-one, (i.e., each tuple in one relation finds a corresponding matching tuple in the other relation) binary join. The size of each tuple is about 1K bytes and the data values are uniformly distributed. However, 20% of the data region is considered hot region following the 80-20 rule. Relation S never changes throughout the whole experiment process; it stands for static relations, such as company profile, staff profile, etc. Relation R is a dynamic data source where the data is modified frequently, e.g., the stock market. It consists of a unique *Identity* and a *Change Ratio* (define the value change over two subsequent sessions).The *Change Ratio* follows a normal distribution with a mean value of 0 and standard deviation of 1.0. We note that a hot region need not be having a high *Change Ratio*, and vice versa, e.g., high trading volume in stock may not always cause high variations of trading price and vice versa. We denote by *Modified Ratio* the percentage of tuples being modified at any given time. We set *Modified Ratio* as 10%; 80% of the modified data comes from the hot region[3]. Queries computation are directly against the data changes. This is a fair assumption since both methods (i.e., existing method and our proposed methods) are working on the same dataset and we are only interested in the relative performance gain.

4.2.2 Query

In our experiments, we use three types of queries to represent the possible queries that users may submit to a CQS. We categorize queries into *Simple Selection Query*, *Range Selection Query* and *Join Query*.

```
Simple Selection Query:
Example: Notify me when Intel stock price changes
Range Selection Query:
Example: Notify me all the stocks whose price changed more than 5%
```

[3]This assumption is logical; since a region is hot, it will be modified more frequent than others

```
Join Query:
Example: Notify me of all stocks whose price changed more than 5%
         and their related company profile.
Note: Assuming stock info and company profile stored in different relation.
```

Simple Selection Query is a group of queries that have the same expression signature on the equal selection predicate on *Identity*. *Range Selection Query* is a group of queries that have the same expression signature on range selection predicate on *Change Ratio.Join Query* is a class of queries that contain expression signature for both selection and join operators. Selection operators are pushed down under join operators.

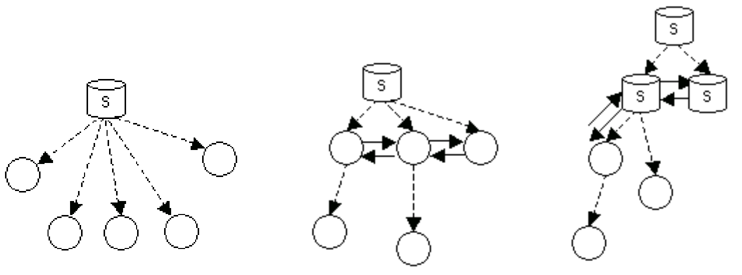

(a) Centralized CQS. (b) CQ-Buddy without SR. (c) CQ-Buddy with SR.

Figure 7: System configurations with can-shape represent data source provider. Dashed arrow lines represent data stream flow, and solid lines local similar queries optimization.

4.3 Experimental Results

In this section, we report the results of our evaluations on a wide range of configurations.

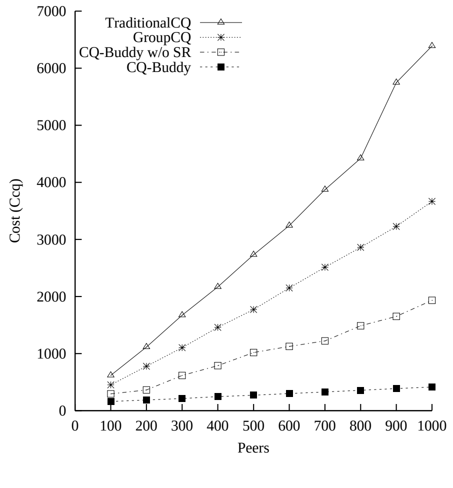
4.3.1 CQ-Buddy vs. Traditional CQS

In the first experiment, we compare the performance of existing CQS with CQ-Buddy. Existing CQS can generally be classified into two types. In the first type of CQS, queries are shared (grouped sharing)[1, 19] techniques. In the second type of CQS, queries are not shared. [9]. We refer to the former CQS as *GroupCQ* and the latter as *TraditionalCQ*. In addition, we note that there are two configurations for CQ-Buddy; CQ-Buddy that employs data stream reallocation (SR) strategy and CQ-Buddy without SR strategy (see Figure 7(c) and Figure 7(b)).

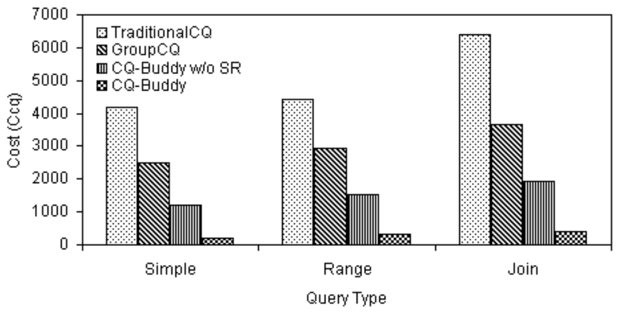
We employ the network topology as shown in Figure 7(a) for *GroupCQ* and *TraditionalCQ* setup, where there is only one data stream provider and a large number of autonomous CQ peers. Similarly for *CQ-Buddy w/o SR*, there is only one data stream provider in the network. In contrast to these configurations, data steams are reallocated to 10% of the delegated peers in the CQ-Buddy configuration. For example, if there are 100 peers requesting stream from CQD, SR strategy will randomly choose 10 of the peers as intermediate stream peers.

Each CQ peer consists of 10 basic queries, and another query set consisting of 50 queries following the 80-20 rule (i.e., 80% of the queries access a hot region representing 20% of the entire data stream) is introduced into the system at runtime. Queries are submitted to the CQD. As in existing single CQS, a new incoming query is checked to determine whether it can be shared with one of the basic queries. If the incoming queries cannot be shared, they are processed separately from the existing queries. Hence, we control this property with $\alpha = 0.4$. However, since there is no query-sharing for the *TraditionalCQ* system, we turn this feature off for *TraditionalCQ* in the experiments. If there are more than one CQD, i.e., CQ-Buddy, the *Random* (refer Section 3.4) CQD selection policy will be used, where each CQD has a fair chance of being selected.

We vary the number of peers that request stream from CQD from 100 to 1000 peers. Three types of queries are introduced in the experiments: *Simple Selection Query*, *Range Selection Query* and *Join Query*. Figure 8 shows the results of the experiments.



(a) Evaluation on vary number of peers.



(b) Cost C_{cq} for different query types on 1000 peers.

Figure 8: The effects of number of request peers and query types on different CQS.

From Figure 8(a), we observe that *TraditionalCQ* is the worst compared to other CQS. This is expected since there is no sharing of similar queries. Hence, each of the incoming query generates a new independent process and bundles the system. On the other hand, in a *GroupCQ* system, similar queries can be grouped together to share computational power. *CQ-Buddy w/o SR* behaves like a *GroupCQ* system since both of them only consider one data stream provider. However, unlike *GroupCQ* system which is autonomous and where all queries grouping and sharing strategy are based on its local decision, *CQ-Buddy w/o SR* is able to share queries among peers and reduce the number of queries involved in the CQD. This is reflected in the graph where *CQ-Buddy w/o SR* outperforms *GroupCQ* system. However, since there is only one single CQD, it eventually becomes a bottleneck. *CQ-Buddy*, which employs the SR strategy, can significantly reduce the load on CQD to provide lower cost, C_{cq}, compared to other CQS. Figure 8(b) shows the C_{cq} based on different queries. From the results, we can observe that through sharing queries among peers and data stream reallocation, there is significant performance improvement. The gain is more obvious for join queries than selection queries because join queries are computationally more expensive than selection queries. CQ-Buddy performed the best in all the three different query types. This is because CQ-Buddy employs internal query sharing, external query sharing and stream reallocation, and these reduce the computation needed for each query and eliminate the bottleneck.

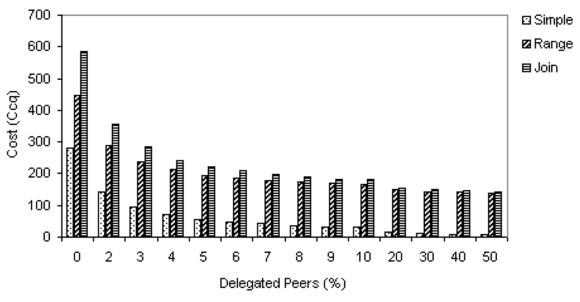
4.3.2 Evaluation of Stream Reallocation Strategy

When a peer is overloaded, stream reallocation proves to be an efficient technique for reducing the load at the data stream provider, thus achieving load balancing among the peers. However, one interesting issue needs to be addressed; "*what is the optimal number of peers that should be used for stream reallocation*?". If this is set too high, it may waste resources without any significant performance gain. On the other hand, if it is set too low, issues relating to overloading may still exist. In most systems, it may be easy to collect statistics on the average number of peers' connection and queries that are submitted. We hypothesize this in the experiment, and make use of a control parameter, *Freq*. *Freq* is an integer number indicating the average number of queries that would be submitted by a peer in a second. In addition, we introduce the following metric, *Cost Saving Ratio*, to measure the results. The *Cost Saving Ratio* is defined as:

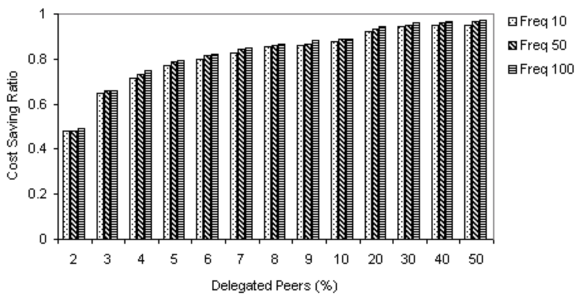
$$Cost\ Saving\ Ratio = \frac{wcost - cost(p)}{wcost}$$

where *wcost* is the cost of answering the query in the worst case, and *cost*(p) is the cost of answering the similar query with p number of peer stream reallocation. For the worst case scenario, we assume one single CQD handling all stream request (Figure 7(a)) and no queries sharing either internally or externally (i.e., among peers).

In the following experiments, we will make use of configurations similar to the previous experiments. At any time, there will be 100 peers in the queue that are waiting to be processed. In Figure 9(a), we vary the percentage of stream delegation peers from 1% up to 50% ("0" means that there is no stream reallocation, and all streams originate from a single CQD). We compare the C_{cq} for different types of queries. The $Freq$ is set to 100, i.e., each peer will submit 100 queries to the CQD. From Figure 9(a), when the percentages of delegated peers increase, the cost for processing



(a) Cost for different query vs delegation peers (%).



(b) Cost saving ratio vs. delegation peers (%).

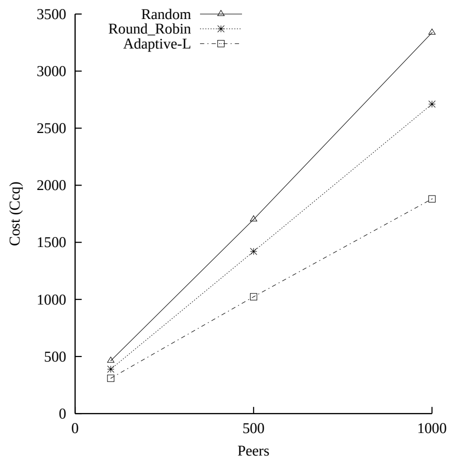
Figure 9: Load vs. number of delegation peers.

queries decreases accordingly. This is because the load has been distributed among intermediate peers. The performance gain can be observed for all the three different types of queries. However, we observe that when more than 10% of the peers play the role of intermediate peers, the performance gain is not as significant. For example, when we increase the percentages of intermediate peers from 10% to 20%; *Join Query* gains less than 4% of cost saving, and *Simple Selection Query* gains only another 0.01% of cost saving. The reason is that the load of each peer decreases when more peers are delegated for doing similar processing. There will come a stage when increasing the number of delegated peers will not be feasible since each of them may be under-loaded, and the performance gain will not be significant. Based on the observation, we hypothesize that 10% of delegated peers are enough to achieve an optimum, whereby increasing the number of delegated node will not lead to further significant performance improvement.

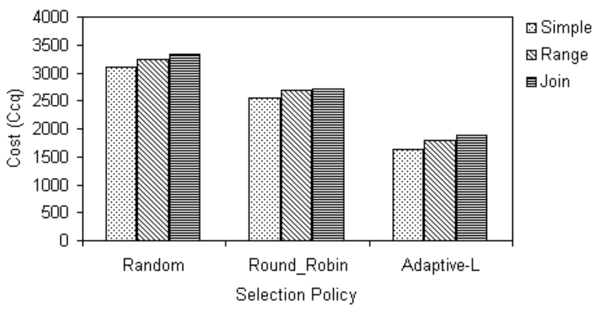
In order to verify this hypothesis, we increase the $Freq$ from 10 to 100, which can be seen as increasing the load for a CQD. The results are presented in Figure 9(b). We evaluate the *Cost Saving Ratio*, which compares the percentage of stream delegation peers. In addition, we consider *join queries*, as it is the most computationally expensive queries amongst the three types of queries which we considered earlier. From the result, we observe that the cost saving will be minimum when more than 10% delegation peers are used.

4.3.3 Evaluation of Stream Selection Policy

The next experiment evaluates the performance of the *Random*, *Round Robin* and *Adaptive-L* stream selection strategies. From the result of Experiment 4.3.1, CQ-Buddy has been shown to be a promising technique. Thus far, we have assumed that all peers have equivalent processing power and equivalent resources. However, this may not hold in practice. In this experiment, we introduce a "CPU" parameter where $CPU \in \{0, 1\}$. This parameter is indicative of the computing power range of a peer. Based on the zipfian distribution with $\theta = 0.4$, the highest range computing power is around 10 times of the lowest range. First, we fix the query type to *Join Query* and vary the number of peers from 100 to 1000. As in the previous experiment, each peer introduces 50 queries on runtime and 10% of intermediate stream providers exist in the network. The results are shown in Figure 10.



(a) Cost C_{cq} vs peers.



(b) Cost C_{cq} vs. stream selection policies.

Figure 10: Effect of different stream selection policy in heterogeneous peers environment.

As shown in Figure 10(a), we note that when the number of peers involved in the system increases, the C_{cq} to complete each set of queries from a peer also increases. The cost increase is almost linear for all three approaches since the load increases when more peers submit their queries. The *Random* policy is the worst among the three policies because in an environment with a large number of medium/low range peers, the probability that a medium/low range peer being selected is the highest. On the other hand, the *Round Robin* policy ensures that the most powerful range peers are selected at a fixed rate. However, it cannot avoid low range peers being selected as in the *Random* policy. This is the reason *Adaptive-L* outperforms *Random* and *Round Robin* since it is aware of the capabilities of each peer, and can make a precise selection of peers with the highest capabilities and lowest load.

More interesting is Figure 10(b) where we fix the number of peers at 1000, and evaluate different query types on selection policies. Again, *Adaptive-L* outperforms the other two approaches. In addition, we note that *Adaptive-L* is not sensitive to the different type of queries.

5 Related Works

Continuous queries (CQ) are used extensively as a useful tool for the monitoring of updated information. CQ is a persistent query that notifies the user when the source of data changes or becomes available. The concept of continuous queries was first introduced by Terry et al. [18] who implemented timer-based continuous queries over append-only database. The approach is too restricted, i.e., it is confined to append-only systems and disallows deletions and modifications. Hence it is not adaptable to dynamic environments such as those found in a distributed or P2P context.

There has been considerable research done in continuous queries processing. More recently, there are several CQ systems developed or proposed for monitoring and delivering information on the Internet. OpenCQ [9] employs an SQL like query language and runs on top of a distributed information mediation system that integrates heterogeneous data sources. The NiagaraCQ system [1] and Xyleme system allow the monitoring of XML documents found on the web. In addition, both CACQ [10] and AdaptiveCQ [19] take note of the need for adaptivity and propose techniques based on the eddies mechanism to facilitate adaptive continuous query processing.

All the systems mentioned above are fundamentally different from CQ-Buddy in several ways. First, most of these existing systems utilize a centralized approach in which the server performs the processing and treat the clients as simply receiving and presenting the information to the end-user. This is typical of a client-server approach. For example NiagaraCQ and TriggerMan [6] explore the similarity among large number of queries and use group optimization to achieve system scalability. Client nodes are treated as simple input/output with very limited participation, i.e., the client sends a CQ query to the CQ server, and waits for results. This contradicts with the P2P principle of information sharing and wastage of potential resources available at the clients' end. CQ-Buddy on the hand, explores the potentials of each participant in the network based on P2P technologies.

The requirements of our system match the characteristics of the P2P technology perfectly. In a pure P2P environment there are no global services, resource or schema control. P2P systems, like Napster [11], Gnutella [4] , ICQ [7] and SETIHome provide for content sharing, communication and sharing of computational power. An evaluation of P2P systems can be found in [21]. These systems, they are limited to transferring content at the object level and cannot support the execution of complex queries across multiple sources, nor use intermediate results in order to answer consecutive queries.

Recently, the peer-to-peer (P2P) computing model has been increasingly deployed for a wide variety of applications in the area of database management, including data mining, replica placement, resource trading, data management and file sharing (see [14, 15]). *Piazza* [5] is the first system to deal with database management issues in P2P systems. It provides a scheme for the indexing of views, mechanisms for distributing an index in P2P network and the exploitation of materialized views. Bernstein et al. [16] propose the Local Relational Model (LRM) to solve data management issues in P2P environment. Each peer in the P2P network consists of a local relational database, with a set of acquaintances that define the network topology. For each acquaintance link, domain relations define translation rules between data items, and coordination formulas define semantic dependencies between the two databases. PeerDB [13] is a P2P-based system for distributed data management and sharing. It supports share data without a shared global schema by employed Information Retrieval based approach. These systems focus mainly on data placement and management problems, and are fundamentally different from CQ-Buddy, as CQ-Buddy is focused on data stream optimization in the P2P network.

CQ-Buddy builds on and extends BestPeer [12] for CQ applications. Briefly, BestPeer is a generic P2P system designed to serve as a platform to develop P2P applications easily and efficiently. It has the following features: (i) it employs mobile agents; (ii) it shares data at a finer granularity as well as computational power; (iii) it can dynamically reconfigure the BestPeer network so that a node is always directly connected to peers that provide the best service; (iv) It employs a set of location independent global name lookup (LIGLO) servers to uniquely recognize nodes whose IP addresses may change as a result of frequent disconnection and reconnection.

6 Conclusion

In this paper, we have presented a novel distributed system that processes continuous queries using Peer-to-Peer technology, called CQ-Buddy. We have shown that CQ-Buddy is able to provide significant performance gains by sharing continuous queries with other peers in an efficient and effective manner. The system is fully distributed and highly scalable as there is no single-point failure and single-source bottleneck. CQ-Buddy network is dynamic and it does not require any specific structure. It also does not require any predictable pattern of participation from the peers. Peers in the CQ-Buddy network also turn their heterogeneity to their advantage, so that “weaker” peers such as PDAs and other mobile devices are helped by “stronger” peers for complex queries processing.

As shown in the experiment evaluation, CQ-Buddy achieves significant performance gains with respect to traditional CQ systems. This is accomplished by (i) two-phase query optimization techniques that share queries internally and externally; (ii) the reallocation of single data stream to a number of intermediate peers in order to eliminate the single source bottleneck; and (iii) the stream selection policy which makes precise selections based on the current load and capabilities of peers.

Acknowledgements

We would like to thank Beng Chin Ooi, Kian Lee Tan and Ao Ying Zhou for their insight and discussions on the project, and their comments on the paper. The project was in part supported by the NSTB/MOE research grant RP960668.

References

[1] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, 2000.

[2] M. Cherniack, M. Franklin, and S. Zdonik. Expressing user profiles for data recharging. In *In IEEE Personal Communications*, pages 6–13, 2001.

[3] Freenet Home Page. *http://freenet.sourceforge.com/*.

[4] Gnutella Development Home Page. *http://gnutella.wego.com/*.

[5] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *WebDB Workshop on Databases and the Web*, 2001.

[6] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Intl. Conf. on Data Engineering (ICDE)*, pages 266–275, 1999.

[7] ICQ Home Page. *http://www.icq.com/*.

[8] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias., and K. L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 25–36, 2002.

[9] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. In *IEEE Knowledge and Data Engineering, Special Issue on Web Technology*, volume 11, No.4, pages 610–628, 1999.

[10] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, Madison, USA, 2002.

[11] Napster Home Page. *http://www.napster.com/*.

[12] W.S. Ng, B.C. Ooi, and K.L. Tan. Bestpeer: A self-configurable peer-to-peer system. In *Intl. Conf. on Data Engineering (Poster) (ICDE)*, page 272, 2002.

[13] W.S. Ng, B.C. Ooi, K.L. Tan, and A.Y. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *Intl. Conf. on Data Engineering (ICDE)*, 2003.

[14] International Workshop on P2P Systems. *http://www.cs.rice.edu/Conferences/IPTPS02/*. 2002.

[15] B.C. Ooi, K.L. Tan, H.J. Lu, and A.Y. Zhou. P2p: Harnessing and riding on peers. In *The 19th National Conference on Data Bases*, August 2002.

[16] A. B. Philip, G. Fausto, K. Anastasios, M. John, S. Luciano, and Z. Ilya. Data management for peer-to-peer computing: A vision. In *WebDB Workshop on Databases and the Web*, 2002.

[17] A. Rowstron and P. Druschel. Past: A large scale persistent peer-to-peer storage utility. In *Workshop on Hot Topics in Operating Systems (HotOS)*, November 2001.

[18] D. Terry, D. Holdberg, D. Nichols, and B. Oki. Continuous queries over append-only database. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, 1992.

[19] W. H. Tok and S. Bressan. Efficient and adaptive processing of multiple continuous queries. In *Intl. Conf. on Extending Database Technology (EDBT)*, pages 25–27, Prague, Italy, 2002.

[20] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[21] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Intl. Conf. on Very Large Data Bases (VLDB)*, pages 561–570, 2001.