Technical Report NUS-CS01-03

Contact Author: Wee Siong Ng

Affiliation & contact address:
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543

Email: ngws@comp.nus.edu.sg

Phone: (65)-6874-4774

Fax: (65)-6779-4580

Title: Efficient Distributed Continuous Query Processing using Peers

Authors: Wee Siong Ng, Yanfeng Shu, Wee Hyong Tok

Efficient Distributed Continuous Query Processing using Peers

Wee Siong Ng Yanfeng Shu Wee Hyong Tok

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 17543
{ngws,shuyanfe,tokwh}@comp.nus.edu.sg

Abstract

In this paper, we present the design and evaluation of CQ-Buddy, a peer-to-peer (p2p) continuous query (CQ) processing system that is distributed, and highly-scalable. CQ-Buddy exploits the differences in capabilities (processing and memory) of peers and load-balances the tasks across powerful and weak peers. Our main contributions are as follows: First, CQ-Buddy introduces the notion of pervasive continuous queries to tackle the frequent disconnected problems common in a peer-to-peer environment. Second, CQ-Buddy allows for inter-sharing and intra-sharing in the processing of continuous queries amongst peers. Third, CQ-Buddy peers perform query-centric load balancing for overloaded data source providers by acting as proxies. We have conducted extensive studies to evaluate CQ-Buddy's performance. Our results show that CQ-Buddy is highly scalable, and is able to process continuous queries in an effective and efficient manner.

1 Introduction

Peer-to-Peer (P2P) technology, also called peer computing, is emerging as a new paradigm that is now viewed as a potential technology that could re-architect distributed architectures. 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. The nodes, which can be both a data consumer and provider, may join and leave the P2P network at any time, resulting in a truly dynamic and ad-hoc environment. Furthermore, the nodes could have idle resources (processing and memory) which can be exploited by other nodes in a secured manner to help process a portion of a distributed task.

Continuous queries 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:

Monitor the Singapore Stock Exchange indefintiely,
notify me when Straits Time Index
current value > 1300

Figure 1: Example of a CQ query.

In the literature on continuous queries, much of the existing work focuses on efficiently handling the processing of a large number of continuous queries by exploiting similarity in the queries, and subsuming a new incoming query into existing queries groups [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 single 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 providers themselves could be potential bottlenecks. A popular data provider may be easily overwhelmed by requests and consequentially delay the response of a CQS. Third, multiple continuous query systems do not share computations, and each function autonomously and is concerned with the efficient and effecti! ve execution of continuous queries within

itself. Multiple CQSs also do not share any query processing task. In short, much of the work performed by individual CQSs is duplicated. Futhermore, resources at some CQSs could be under-utilized. For example, a large number of CQSs may be accessing the same data provider, thus overloading the data provider and causing it to become a bottleneck.

In this paper, we present the design and evaluation of CQ-Buddy, a peer-to-peer (p2p) continuous query (CQ) processing system that is distributed and highly-scalable. CQ-Buddy exploits the differences in capabilities (processing and memory) of peers and load-balances the tasks across powerful and weak peers. Furthermore, CQ-Buddy introduces the notion of pervasive continuous queries, to allow complex continuous queries to be processed by other buddy peers when a peer gets disconnected. Second, CQ-Buddy allows for inter-sharing and intra-sharing in the processing of continuous queries amongst peers. In CQ-Buddy, intra-sharing of queries is achieved by grouping similar queries and processing them within the continuous query mechanism of the node, whereas inter-sharing is achieved when multiple CQ-Buddies help one another by processing continuous queries in a distributed manner. Third, we note that data providers may be overwhelemed with queries, and may become a bottleneck.CQ-Buddy peers help to alleviate overloaded data providers by performing query-centric load balancing for overloaded data source providers by acting as proxies.

The rest of this paper is organized as follows. In the next section, we discuss how peers can be harnessed for continuous query processing. We discuss issues that are prevalent when applying P2P to the domain of continuous query processing, and we present solutions. Section 3 provides a formal model for identifying similarities between queries. Section 4 gives an overview of the design and features of CQ-Buddy. In Section 5, we present an extensive experimental study to evaluate CQ-Buddy. In Section 6, we review related works, and finally, we conclude in Section 7 with directions for future work.

2 Towards P2P Continuous Query Processing

In this section, we first provide scenarios on distributed continuous query processing on multiple sites. Next, we look at the features of P2P systems and provide examples on how P2P technology can be used to process continuous queries in a distributed manner. We also look at how peers can help and complement each other in processing queries and perform load balancing. This will also serve to motivate the need for continuous query processing using P2P technology. For this purpose, we shall refer to a node in the distributed P2P network as a *peer*.

2.1 Duplicate Processing of Similar Queries

Most existing continuous query systems [17, 8, 1, 9, 18] are designed to process continuous queries in an efficient manner at a single-site. In a network where there is a large number of computers (nodes), each CQS executing on each computer would process continuous queries independently. There is therefore a large possibility of duplicate processing of continuous queries in a network.

However, if the multiple CQSs executing at various peers could cooperate and "help" each other in processing the queries, the amount of duplicate processing can be significantly reduced, and thus improving overall system responsiveness. The grouping of similar queries to allow for sharing of computation has in fact been the focus of many CQSs. With increased opportunities for sharing, query processing can be further optimized holistically across all CQSs. Contrast this with a single CQS, where there are relatievly lesser opportunities for similar queries.

2.2 Data Providers - Bottlenecks?

When a large number of peers access the same data source, the data provider itself becomes a bottleneck. Most of the existing work focuses on tackling adaptive query processing at the CQS end, but not at the data provider end. However, the data providers themselves may be overloaded by requests from multiple CQSs and hence their performance suffers. In our model, we consider two different configurations for data providers.

2.2.1 Data Provider with Multiple Nodes

In the first scenario, the data provider consists of multiple nodes, with each node providing the same set of data. The peers accessing the data providers are aware of the multiple data providers, and uses a selection policy to determine which data provider node would service a request. (Refer to Section 4.3.4)

2.2.2 Peers as Proxies

In the second scenario, the data provider consists of a single node. The node maintains a list of neighbouring peers which it can delegate as *proxy peers*. Proxy peers fetch data on behalf of other peers, which must otherwise access the data provider node themselves. This cuts down the number of concurrent requests to the data provider node. As the load of the data provider node reduces significantly, the overall responsiveness of the system improves.

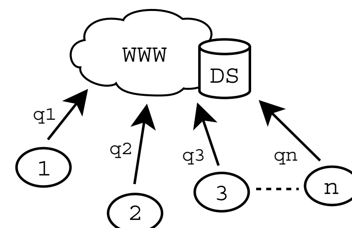
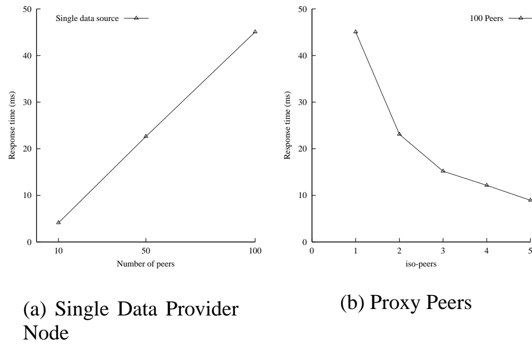


Figure 2: Multiple peers accessing a popular data source.

Let us consider an example. In Figure 2, we have mutiple peers each issuing continuous queries to a popular data provider. The data provider node 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 simple experiment to validate this example. In the first experiment, we create a total of 100 peers (varies from 10 to 100). Each peer submits 50 queries on runtime to a CQS. In the first set of experiment, there is a single data provider node servicing the requests from the multiple peers. The average response time of peers (see Figure 3(a)) is recorded.

In the second experiment, we allow the data provider node to delagate several proxy peers to service the requests. Queries are submitted to these peers in a random manner. Figure 3(b) shows that the response time improves significantly. Thus, we can observe that by introducing proxy peers, we are able to improve the overall responsiveness of the system.



(a) Single Data Provider Node

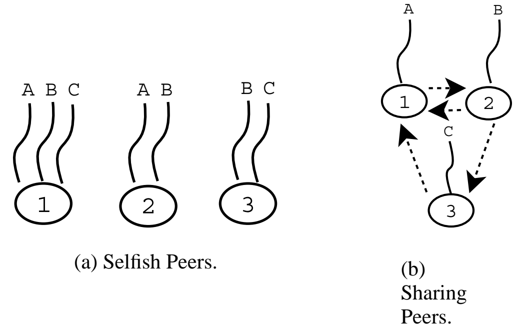
(b) Proxy Peers

Figure 3: Experiment to show the advantages of introducing intermediate proxies

2.3 Resource Sharing Strategies

P2P technology facilitates the sharing of data and computing resources. Intuitively, if we can harness peers in a P2P network to service continuous queries, there is immense potential for enhancing the reliability and performance of all the CQS pariticipating in the P2P network. 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. Let us now consider the scenario 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. Intuitively, the workload can thus be evenly distributed amongst the peers, instead of having several single overloaded peers.

Furthermore, peers may not have equal resources. Peers can be running on a variety of devices, ranging from a Personal Digital Assistants (PDA) to a laptop or a desktop.



(a) Selfish Peers.

(b) Sharing Peers.

Figure 4: Peers' relation.

The basic idea behind CQ-Buddy is to allow peers that are weaker than its neighboring peers to seek help from buddy peers in processing similar continuous queries.

2.4 Frequent Connection/Disconnection of Peers

Before leaving this section, let us look at an example that motiviates the need for pervasiveness in continuous queries.

Let us consider a business traveller, who wishes to perform a long running computation (based on a complex financial model) on real-time updates of the NYSE Composite index. Furthermore, he needs the computed results upon arrival at the destination. When the traveller boards the plane, his PDA is disconnected from the network of peers. However, prior to disconnecting, his peer software asks for help from its buddy peers to perform the query. When he arrives at his destination, he powers up his PDA and immediately, the buddy peers provide him (rather his PDA) with the computed results from the complex, long running function that has been applied to an underlying continuous data stream (i.e. from the New York Stock Exchange).

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*. It is useful when a peer can leverage on other buddy peers to process a long running processing during its absence from the network.

3 A Formal Model for Similar Queries

In this section, we present a formal model for expressing similar queries, and we make use of the model for detecting similarities in CQ-Buddy. Table 1 provides the terms used in the model.

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*

Terms	Description
PredSim(Q_1, Q_2)	Similarity between two predicates Q_1, Q_2
ProjSim(Q_1, Q_2)	Similarity between two project attributes Q_1, Q_2
DSSim(Q_1, Q_2)	Similarity between two specified data sources Q_1, Q_2
QuerySim(Q_1, Q_2)	Similarity between two queries Q_1, Q_2

Table 1: Terms of Reference

$$Col_1 = Col_2 \text{ and } \circ_1 = \circ_2$$

We refer to the above definition as a strong *similarity. In addition, we also relax the constraints on* $\circ_1$ *and* $\circ_2$*. Two selection have a* weak *similarity if and only if*

$$Col_1 = Col_2$$

(Note: CQ-Buddy supports both weak and strong similarity.)

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 unique 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

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 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*.*

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 and a system-defined threshold should be used to decide whether to treat the two queries as similar or dissimilar.*

4 CQ-Buddy: A Distributed CQS Using Peer Technology

In this section, we shall present CQ-Buddy, a peer-to-peer (P2P) continuous query system. We shall first look at the CQ-Buddy network and the architecture of a CQ-Buddy node. 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.

4.1 CQ-Buddy Network

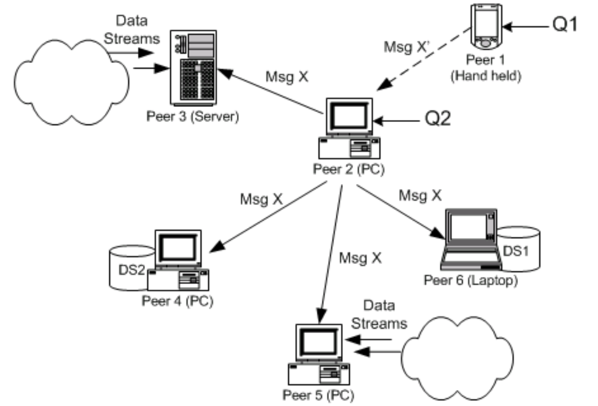


Figure 5: Overview of CQ-Buddy Network.

CQ-Buddy is a P2P-enabled distributed CQS. In CQ-Buddy, we distinguish between two different roles of peers. First, peers can act as *proxies* to data providers and help to reduce the number of concurrent requests to the data provider nodes (e.g Peer 2 and 5 in Figure 5). Second,

each peer implements a continuous query system that co-operatively interacts with other peers to process continuous queries (e.g. Peer 3 to 6 in Figure 5).

Let us consider the case where a new query Q2 = *select * from sti.stream where Stock.symbol = 'Creative 50' or Stock.symbol = 'SIA'* is submitted to Peer 2. *sti.stream* retrieves the stock indexes from the Straits Time Index which is provided by the Singapore Stock Exchange. All incoming queries that are submitted by the user to a peer are first optimized. If a query is similar to one of the existing queries, it is subsumed into an existing query group.

If the query is not similar, the peer could either process it by itself or ask another peer (ie CQ-Buddy) which is already processing a similar query to help. In the second option, the peer sends a "help" message with the newly arrived query to other peers to see whether they are already processing a similar query. 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. Stronger peers, with more resources (i.e. processing and memory) help weaker peer in processing continuous queries. Note that the objective is to locate peers which currently handle *similar processes* (i.e., monitoring data source sti.stream with projection attributes Stock.symbol), so no exact match of projection attributes is necessary.

When Peer 2 sends a "help" message to other peers, it has no advance knowledge of the number of peers that will respond. Instead, it relies on a predefined threshold (e.g., stop when 2 peers 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.

When a peer receives a request, it may either handle the query if it has 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. In order to break potential message loops, each peer keeps a queue of the recent messages and rejects the ones that have been processed before. Peers which are able to handle the query (i.e. able to merge the incoming query into their existing process groups) 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.

[1]CQ-Buddy is built on top of BestPeer [11]. BPID is a global identity used in BestPeer to uniquely identify different peers and their respective location in the dynamic network.

4.2 Architecture of a CQ-Buddy Node

Let us consider the architecture of a CQ-Buddy node. Figure 6 depicts the architecture of an autonomous peer in

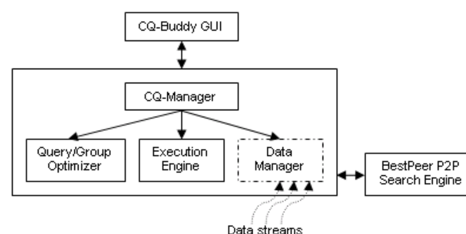


Figure 6: Architecture of a peer.

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. An incoming query will first be optimized internally with a peer. The queries or grouped queries that cannot be subsumed into an exisiting query will then be used as input for the P2P search engine to locate the other peers that can handle the queries. Note that the Data Manager module may not be operational in a peer, since it simply consumes data provided by the data provider and act as an intermediate proxy for other peers if there is a need for load balancing. The data manager in a peer monitors the data sources (i.e. a flat file, DBMS or data streams from devices in the network). Here, we assume that the data are read-only, and that there is an implicit time attribute tagged to all data. CQ-Manager invokes the execution engine to evaluate the installed continuous queries. Second, the CQ-Manager orchestrate the queries that are processed by other peers, and handles the return of the results to the CQ-Buddy GUI.

4.3 Optimization of CQ-Buddy

Let us now consider the features of CQ-Buddy, and how it can exploits the differences in capabilities of peers and load-balances the tasks across powerful and weak peers.

4.3.1 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 5, we perform an extensive study on the effectiveness of these two proposed strategies.

4.3.2 Support for Pervasive Queries

CQ-Buddy supports a novel class of continuous queries, called *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 [2, 3, 10, 16]. For example, a mobile device such as a 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 users in terms of access to requested information. Consider another query Q1 that is defined as Q1 = *select computeFinancialModel(real_time_indexes) 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 is potentially long running due to the need to perform a complex computation (i.e. due to the *computeFinancialModel* function) and the buddy peer which handles the query will help to store the result for 30 minutes. Peer 1 submits the pervasive query Q1 to Peer 2, and disconnects from the P2P network after receiving an acknowledgment from Peer 2 (denoted by dash line). Peer 2 handles the query if it is a normal query in the manner as described previously. However, Peer 1 will store the results for the long running operation and return the results to Peer 1 when Peer 1 reconnects to the P2P network. In a P2P network, the frequent disconnection of peers to the network can interrupt the processing of continuous query that may require complex computation. CQ-Buddy solves this problem of frequent disconnection of peers by introducing *pervasive CQ*.

4.3.3 Query-Centric Load Balancing for Data Providers

We consider that the main data provider (i.e. a peer in the P2P network) may be overloaded with queries, and thus performance suffers for all CQSs requesting data from the data provider. Each query may request for a specific set of attributes, or a specific range of data values. When there are many queries requesting for a specific set of data values or attributes, we refer to this frequently requested data collectively as a *Hot Region*.

In order to reduce the load on the data provider, the Hot Region is a potential candidate for being delegated to an intermediate proxy peer to handle the queries. This will help offload the burden from the main data provider.

Each peer maintains a list, call *Hot Region List* of data attributes that are often requested by the queries from the various peers. A *Hot Region List* consists of a *Hot Region*, *Hot Count* and a *Window Count*. The list is sorted in descending order based on the value of *Hot Count*.

The *Hot Region* keeps track of the data attributes that are hot. The *Hot Count* is a counter that keeps track of the number of queries requesting for data attribute in the corresponding Hot Region. For each request to a Hot Region, we increase *Hot Count* by 1. However, due to the long-running nature of continuous queries, the effect of a Hot Region may vary across time. Hence, past hotness of a region would influence its current hotness. In order to reduce this effect, we consider dividing time into non-overlapping windows. Within each window, we increase *Window Count* by 1 if there is a request for the corresponding *Hot Region*. At the end of the time interval for the window, we copy the value of *Window Count* to *Hot Count*, and reset *Window Count* to 0. An example of a Hot Region List is presented in Table 2. The values *a,b* denotes data values, and *c-e* denotes a range of values which are requested by the queries.

Hot Region	Hot Count	Window Count
a	15	1
b	13	5
c - e	14	10

Table 2: Example of a Hot Region List

When a data provider is overloaded, it will determine from the Hot Region List, the regions to be delegated to intermediate peers, which would then act as *proxies* to service the request. The selection process is crucial to achieving effective load balancing for the data provider. Noting that each peer may be different in terms of their resources (i.e. processing and memory), only peers which are stable and seldom disconnected from the CQ-Buddy network are considered suitable nodes for delegation as a proxy peer to help offload requests from a single data provider.

4.3.4 Adaptive Selection Policy

Let us consider the effectiveness of different selection policies used in selecting a peer to be used as an intermediate proxy. We consider the problem of selecting a peer amongst a list of m peers, and delegating the peer as a *proxy peer*.

Two naive solutions can be employed. First *Random* policy, in which the probability for selecting any peer is equal to $1/m$. Second, *Round Robin* policy, which works on a rotating basis where one $peer_i$ is selected and used to process queries, and then moved to the back of the list; the next $peer_{1+i}$ is selected and then moved to the end of the list after it has done its job; and so on, until $peer_m$ is selected. However, these policies do not take into consideration giving preference to those peers with the least amount of congestion or workload.

We propose a peer selection algorithm, called *Adaptive-L*, based on a randomized resource allocation technique called lottery scheduling [19] and taking into consideration current load and the processing power of a peer prior to delegating it as a proxy peer to help offload the task of the data provider. The Adaptive-L strategy is a variant of the lottery scheduling technique. The main difference is that instead of using tickets to be used in a lottery, we consider the use of round trip time as a representative measure of the responsivenss of a peer. 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 peer 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 round-trip time for the ping query will be τ will be captured (Ref:1). The round-trip time will then be normalized to an internal scale and this will be used to compute the ticket volume $v(O)$ using the function τ (Ref:2),

If the peer O_{oid} exists in the local cache, the new ticket volume will be combined by getting the a verage value of the new volume value and cache volume value.

Finally, *generateToken* step (Ref:3) is a random function that generates a token in the range of the total sum of ticket volume. This is used to determine the peer that will be selected as a candidate to process the query.

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

```
Data   : a candidate list ω = (O_oid1, O_oid2, ..., O_oidl) of response
         peers whose are able to process the query.
Result : A selected O_oidx object.
begin
     nω ⟵ ∅, τ ⟵ 0
     for i ⟵ O_oidi ∈ ω do
1        τ = roundTrip(O_oidi)
         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_oidi) = ticketVolume(τ)
         if vc(O_oidi) ≠ nil then
             newVolume = (v(O_oidi) + vc(O_oidi))/2
         else
             newVolume = v(O_oidi)
         nω[i] ⟵ new T(newVolume, O_oidi)
     nω ⟵ sorted nω in ascending T.v order
     aω ⟵ ∅, av ⟵ 0
     foreach element e of the nω do
         av ⟵ av ∪ e.v
         aω[i] = new T(av, e.O)
3    token = generateToken(aω[last])
     next ⟵ 0
     while token ≤ aω[next].v do next++
     return aω[next].O
end
```

5 A Performance Study

We have conducted detailed simulation to study the various CQ-Buddy features discussed in the previous sections. In this section, we present our extensive performance evaluation of CQ-Buddy. First, we show the benefits of CQ-Buddy in allowing multiple CQSs in a P2P network to cooperate and help each other. Second, we show how stronger peers can help weaker peers process continuous queries. Third, we consider the various proxy peer selection policies which can help a data provider reduce the number of simultaneous requests being sent to it. Finally, we look at the effects of the number of delegated peers on query response time.

5.1 Experiment Parameters

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 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.

5.1.1 Data Sets

We run our experiments against two different data sets, R and S. Relation R and S consists of 50,000 and 100,000 tuples respectively. 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.

5.1.2 Queries

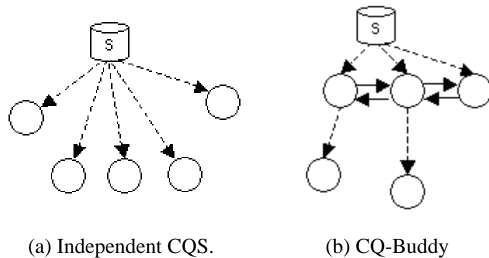
In our experiments, we use three types of queries to represent the possible queries that users may submit to a CQS. We categorize queries into *Simple Selection Query*, *Range Selection Query* and *Join Query*.

```
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%

Join Query:
Example: Notify me of all stocks
         and the company names
         whose prices changed more
         than 5%
Note: Assuming stock info and
      company profile (i.e. name)
      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.

5.2 CQ-Buddy vs. Independent CQS

In the first experiment, we compare the performance of existing CQSs with CQ-Buddy. Existing CQSs can generally be classified into two types. In the first type of CQSs, queries are shared (grouped sharing)[1, 18] techniques. In the second type of CQSs, queries are not shared [8]. We refer to the former CQS type as *GroupCQ* and the latter as *TraditionalCQ*. In the experiment, we shall consider GroupCQ, since the later is able to allow computation for similar queries to be shared and is thus more efficient and effective compared to TraditionalCQ.



(a) Independent CQS. (b) CQ-Buddy

Figure 7: Independent CQS vs CQ-Buddy

Each CQ peer consists of 10 basic queries, and another query set consisting of 10 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 peers.

Similar to the case of a single CQS, a new 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. We set the degree of overlap, for similar queries to be $\alpha = 0.4$. We vary the number of peers from 100 to 1000 peers.

In the GroupCQ case, we make use of several peers each running a CQS, independent of each other. Peers in the GroupCQ case do not interact with each other, and process the continuous queries with no knowledge of the continuous query that are being processed in other CQSs. In the CQ-Buddy case, peers help one another in processing similar queries. We compare the performance of these two cases.

We study the performance of GroupCQ and CQ-Buddy using hree types of queries: *Simple Selection Query*, *Range Selection Query* and *Join Query*. Figure 8 shows the results of the experiments. From Figure 8(a), we note that GroupCQ performs slightly better than CQ-Buddy when the number of peers is small (less than 250). This is due to cost of passing message to explore which other peers can process a similar query. However, when the number of peers increases, it can be observed that CQ-Buddy outperforms GroupCQ. In Figure 8(b) and Figure 8(c), as the nature of the operations get more complex (i.e. join queries), the benefits of being able to cooperatively process similar queries amongst peers become apparent.

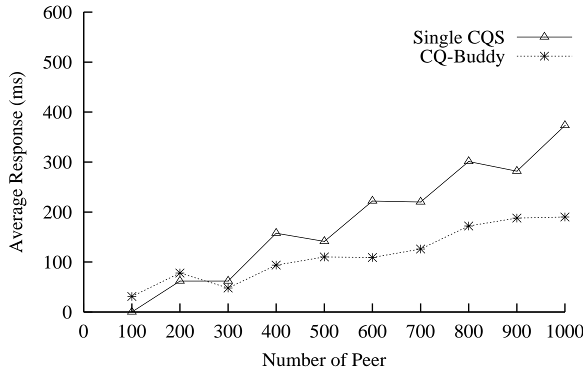
It can be observed from Figure 8 (a)-(c) that when the number of peers is small, the cost of message passing between CQ-Buddy peers dominates, and hence the performance of CQ-Buddy suffers. However, in a large P2P network, the number of peers participating is potentially large, and hence substantial benefits can be reaped from being able to cooperatively process queries.

We also studied the number of messages that are passed between CQ-Buddy peers when processing join queries. Table 3 shows the number of messages that are exchanged between CQ-Buddy peers when identifying CQSs that can help process a similar query. From Table 3, we note that even though the number of messages that are exchanged between CQ-Buddy peers is large, the average response time for CQ-Buddy still outperforms GroupCQ by more than 50%.

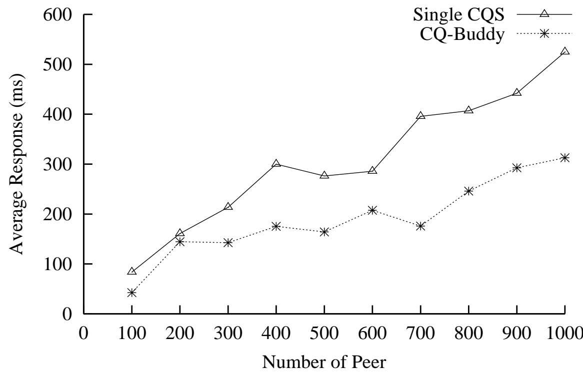
5.3 Weak and Strong Peers

In this experiment, we define a parameter θ, to specify the resources (i.e. processing, resources) of a peer, ranging from {1-10}, where 10 is the most powerful, and it is 10 times more resources than the 1.

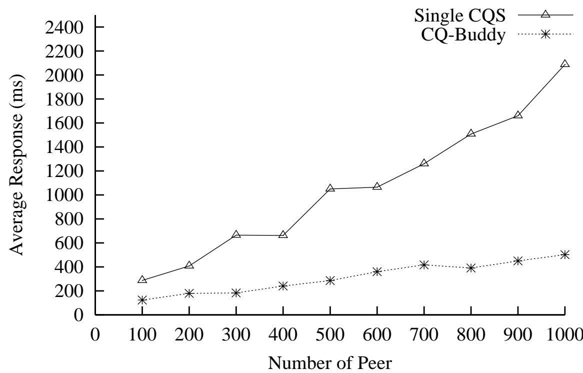
For all the peers (total 1000 peers, each peer sends 10 queries on runtime), we divide them into a weak and a strong group. We define weak as $\theta = \{1 - 4\}$, else strong as $\theta = \{5 - 10\}$ So, for each peer belonging to the weak group, we randomly select $\theta = \{1 - 4\}$. Similarly, for the strong peers, we randomly select θ from the range of $\theta = \{5 - 10\}$.



(a) Selection Queries



(b) Range Queries



(c) Join Queries

Figure 8: Traditional CQS vs CQ-Buddy for different query types

We consider two scenarios depicted as CQS and BCQ in Figure 9. In the first scenario, CQS, we assume that each peer processes queries on its own, without help from stronger peers. When CQS=10%, it means that 10% of weak peers exist in the environment, and weak peers have to perform continuous query processing on their own, without any help from the strong peers. In the second scenario, we consider the case where stronger CQ-Buddy peers help weaker peers in processing similar queries. When BCQ=10%, it means that there are 10% of weak peers, but these peers are helped by strong peers.

	CQ Buddy		Group CQ	
Number of Peers	Total Msgs	Average Response Time (ms)	Total Msgs	Average Response Time (ms)
100	3585	123.93	0	287.72
200	5087	178.98	0	407.49
300	6754	182.72	0	664.82
400	8165	240.72	0	662.28
500	10228	286.81	0	1,249.63
600	11519	360.72	0	1,064.70
700	13094	416.96	0	1,260.41
800	14729	390.76	0	1,508.46
900	16303	449.95	0	1,659.94
1000	17922	502.16	0	2,087.11

Table 3: Comparison of messages during similar query exploration and average response time for GroupCQ and CQ-Buddy

In the experiment, we increase the percentage of weak peers from 10% up to 50% to evaulaute the effect of strong peers helping weak peers in a CQ-Buddy network. Figure 9 shows the performance improvement from the effects of strong peers helping weak peers.

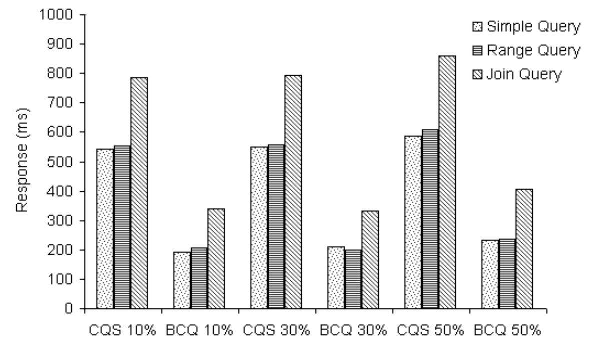


Figure 9: Strong peers helping weak peers

From Figure 9, we can observe that BCQ, where strong peers help weaker peers, consistently outperforms CQS by more than 50%. This is expected, as by noting the disparity in resources available to each peer, substantial performance can be achieved by allowing a more powerful peer to help out in the processing.

5.4 Effect of Proxy Peers Selection Policies

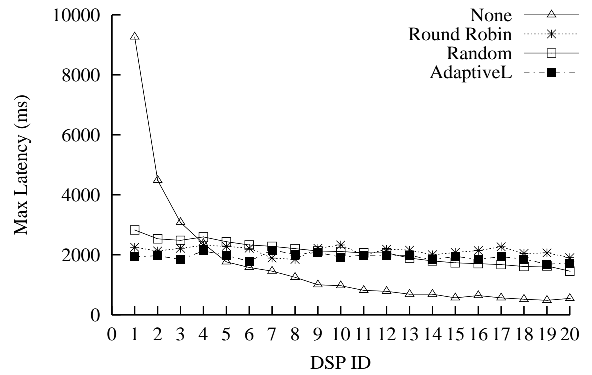
In this experiment, we study the effectiveness of different selection policies used in selecting a peer to be used as an intermediate proxy. We consider the maximum latency on each peer to be the time taken for all queries to be completed on a peer. The x-axis of the graphs in Figure 10 denotes the *id* for each peer that are delegated as *proxy peers*,

and the y-axis denotes the maximum latency for each peer. We assume that in the P2P network, there are 1000 peers, and each peer sends 10 queries to a single data provider. In the first experiment, we assume that all peers that are candidates for selection as *proxy peers* have the same processing capability. In the second experiment, we consider the case where the peers have varying processing capabilities. In both experiments, we use the following strategies: *Random*, *Round Robin* and *Adaptive-L* for selecting a peer as an intermediate proxy.

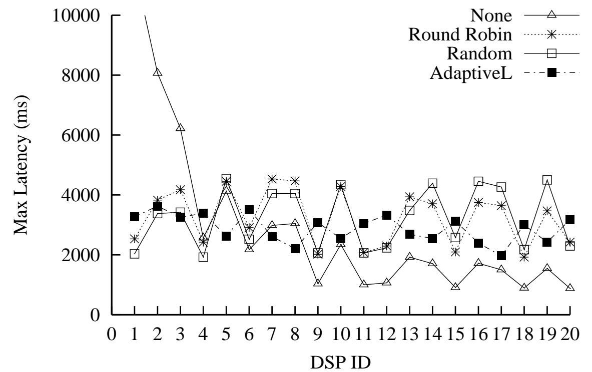
From Figure 10(a), we can observe that when all peers have the same processing capability, the various naive approaches (e.g. Round Robin, Random) are able to ensure that the load (i.e. maximum latency) on each peer is approximately the same.

However, in a large P2P network, it is common that peers have different processing capabilities. In the second experiment, we consider the case where all the peers have different processing capabilities, ranked from 1-10, where 1 is the weakest and 10 is the strongest. All the peers are assigned the ranking randomly. From Figure 10(b), we can see that the maximum latency for each peer fluctuates for the Round Robin and Random policies. When Random Policy is used, the maximum latency for the peer varies from 2000ms (e.g. DSP ID=1) to 4200ms (e.g DSP ID=5). Hence, it is obvious that the load is not evenly distributed amongst the various peers.

However, when AdaptiveL strategy is used, the maximum latency varies from 2000ms to 3200ms for all peers. We can observe from the experiment that the AdaptiveL strategy is more effective in balancing the load by ensuring an evenly distributed maximum latency amongst all the peers.



(a) Load at nodes (equal resources)



(b) Load at nodes (unequal resources)

Figure 10: Effect of node selection policy on load at nodes in a CQ-Buddy network

5.5 Effect of Hot Region Delegation to Peers

In this experiment, we study the effect of the number of peers(delegated as proxy peers to a data provider) on the query response time. In the first experiment, we increase the number of delegated proxy peers. The query response time for three diffferent types of queries are then recorded.

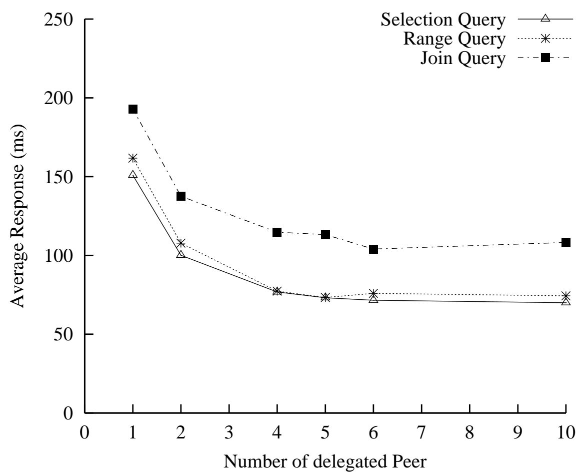
From Figure 11(a), we can observe that as the number of delegated proxy peers increases, the respone time for the various types of queries is reduced. Intuitively, if we reduce the number of concurrent requests to a data provider through the use of proxies (i.e. peers), we are able to service more requests using the intermediate proxy peers. However, due to diminishing returns, increasing the number of delegated proxy peers would not add any benefits to the overall responsiveness of the system. This can be observed in Figure 11(a), the line being flat after five delegations.

In the second experiment, we fix the number of delegated proxy peers at five, and increase the number of peers accessing the fixed number of delegated peers. From Figure 11(b), we can observe that the query response time increases when the number of peers increases. This is expected, since the fixed number of peers is now not able to service so many query requests. Here, there is a need to delegate more proxy peers in order to reduce the load at each proxy peer and improve the overall responsiveness of the system.

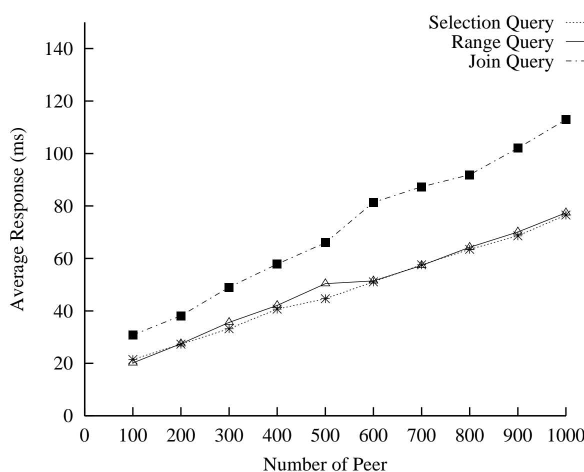
6 Related Works

Continuous queries are used extensively as a useful tool for the monitoring of updated information. The concept of continuous queries was first introduced by Terry et al. [17] 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 [8] 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 sys-



(a) Response Time vs Number of Delegated Peers



(b) Response Time vs Number of Peers (Fixed Number of Delegated Peers)

Figure 11: Effects of number of delgated peers on query response time

tem allow the monitoring of XML documents found on the web. In addition, both CACQ [9] and AdaptiveCQ [18] 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.

The requirements of CQ-Buddy 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 [10], Gnutella [3] , ICQ [6] and SETI@Home provide for content sharing, communication and sharing of computational power. An evaluation of P2P systems can be found in [20]. These systems 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 [13, 14]. *Piazza* [4, 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. [15] propose the Local Relational Model (LRM) to solve data management issues in a 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 [12] is a P2P-based system for distributed data management and sharing. It supports share data without a shared global schema by employing an Information Retrieval based approach. These systems focus mainly on data placement and management problems, and are fundamentally different from CQ-Buddy, which focuses on data stream optimization in the P2P network.

In addition, PeerOLAP [7] looks at P2P data maangement issues in the context of OLAP. PeerOLAP acts as a large distributed cache for OLAP results by exploiting underutilized peers. When a query is issued, the initiating peer decomposes it into chunks, and broadcasts the request for the chunks in a similar fashion as Gnutella. However, unlike Gnutella, PeerOLAP employs a set of heuristics in order to limit the number of peers that are accessed. Missing chunks can be requested from the data warehouse. PeerOLAP also supports adaptive reconfiguration of the network structure, which results in reduced query costs. The system maintains statistics for the most frequently accessed peers. Each peer, at regular intervals, reconsiders its set of neighbors and stays connected to the most beneficial ones.

CQ-Buddy builds on and extends BestPeer [11] 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 a 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.

7 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. The CQ-Buddy network is dynamic and it does not require any specific network structure to be defined. 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 query processing.

As shown in the evaluation, CQ-Buddy achieves significant performance gains with respect to traditional CQ systems. This is accomplished by (i) Allowing inter-sharing and intra-sharing in the processing of continuous queries amongst peers. (ii) Performing query-centric load balancing for overloaded data source providers by allowing peers to act as proxies. In addition, we also note that computations could be long-running and the frequent disconnectivity of a peer from the P2P network could pose a problem. CQ-Buddy solves this by introducing *pervasive continuous queries*, which allows peers to ask buddy peers for help while it gets disconnected from the network, and return at a later time to retrieve the results from the long running computation.

Acknowledgements

We would like to thank Beng Chin Ooi, Kian Lee Tan and AoYing 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] Freenet Home Page. *http://freenet.sourceforge.com/*.

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

[4] 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.

[5] A. Y. Halevy, Z. G. Ives, D. S., and I. Tatarinov. Schema mediation in peer data management systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, Jun 2003.

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

[7] 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*, June 2002.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.