\textbf{\textcircled{Q}Tor:} an incremental approach for collaborative pub/sub systems

Sébastien Dufromentel  
François Lesueur  
Philippe Lamarre

<firstname.lastname@liris.cnrs.fr>
Université de Lyon, CNRS  
INSA-Lyon, LIRIS, UMR5205  
F-69621, France

June 2, 2014

\textbf{Abstract}

The continuous increase of the amount of available data on the Internet and of the number of users, as well as the increasing complexity of the queries –especially the amount of data involved to get an answer– demands the use of efficient, scalable and adaptive systems. In this paper, we propose Q\textcircled{Q}Tor, a query-based publish/subscribe system relying on the mathematical relations between the subscriptions.

While current systems either rely on the data sources to provide computation and network capabilities or require some system nodes to transfer the desired data, Q\textcircled{Q}Tor can be distributed among the subscribers only, which are grouped into communities that collaborate together. We show in this paper that this multi-layered approach provides efficient organizations to distribute the computation and network load while keeping the management mechanisms scalable.

\section{Introduction}

Among current trends in data management systems, the high amount of available data is a challenging problem. Those data may be produced by continuously emitting sources (\textit{e.g.} sensors), or diffused by complex systems with a large number of sources, as can be found on the Web: on both cases, users may express long term subscriptions, continuously executed, to get the newly published results. The difficulty is here to organize such a system, where a lot of subscribers submits queries that may be complex.

To answer this problem, P2P Publish/Subscribe Systems share the work regarding to the subscriptions. In this paper, we propose a user-oriented Pub/Sub system, in which the users are asked to collaborate in order to limit the load. This collaboration is designed to be a win-win situation: the users can benefit from the work done by the others, without having to work out of their own interests.

This kind of organization is made possible by the fact that users may express similar queries, and because some of those queries may be derived from others: several queries in the system may be calculated from the results of others queries as well as from the sources, allowing their users to work together. We propose in this paper an efficient solution to exploit the rewritability relations given by the chosen query language.

In this way, our approach, called Q\textcircled{Q}Tor (for \textit{Query Torrent}), is to split the general problem into smaller parts, by using communities, groups of users that work on the same query. Communities allow to simplify the global problem and to localize modifications when users join or leave the system, in order to achieve a good scalability even for the placement aspects, which are problematic in several existing propositions.

Note that our proposition is compatible with other existing systems tackling the access to data: as the DataBase Management Systems (DBMS) paradigm decouples the access to data (indexes, I/O...) from the execution of queries (query plan optimization...), benefiting from optimizations on both aspects, our Q\textcircled{Q}Tor approach can be completed by the use of a suitable data substrate such as SON, DHT or CDN.
In Section 2, we present the related work, and in Section 3, the theoretical background used by our proposition. In Section 4, we formally define the system and the problem we address. In Section 5, we present our approach, by explaining the general concepts, and we detail the organization workflow in Section 6. The efficiency and scalability of our proposition is shown by the results obtained on this application, given in Section 7.

## 2 Related Work

Our aim is to investigate organizations between multiple queries, or subscriptions, expressed by users, and continuously running in a large scale cooperative setup. This problem is related with continuous queries in databases or data streams management systems, publish/subscribe organizations, distributed stream processing systems. Some of those propositions run in centralized contexts, either with a truly centralized system or with a partition between servers and users; while the others are based on peer-to-peer approaches.

### Query processing

<table>
<thead>
<tr>
<th>Source</th>
<th>Centralized</th>
<th>Isolated</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>NiagaraCQ[8]</td>
<td>Classical Unicast</td>
<td>RoSeS[36]</td>
<td></td>
</tr>
<tr>
<td>TelegraphCQ[7]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STREAM[4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSystem[22]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared filters[28]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System nodes</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>S4[29]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medusa[9]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Users</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicative Multicast[5]</td>
<td></td>
<td>S4[29]</td>
<td></td>
</tr>
<tr>
<td>Delta[23]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Centralization aspects for the concerned propositions

The first line corresponds to fully centralized systems, in which an single source has to perform all the operations (even if, as in RoSeS[36], the users could contribute to the processing of their own query). In the second line, operations are distributed, with more or less collaboration between the nodes, but the system remains a black box in which the user just subscribe and wait for the results. The last line gives fully distributed in which the users are asked to contribute to both processing and diffusion aspects. In the middle column, each node works by its own, while the others allow them to locally optimize a lot of queries, or to collaborate together.

In order to compute a lot of queries with minimal cost, a lot of classical systems rely on the opportunities given by the centralization, which allow to combine different queries. NiagaraCQ[8] or TelegraphCQ[7], for instance, work on the limitation of the I/O operations by the combination of queries having the same “signature”, which mean they perform the same physical operations, even if some parameters change (for instance, comparing the same attribute in the same XML tag, even if the reference value to compare differs). Query plans may also be associated, as in STREAM[4] or Q System[22], by factorizing queries and finding common subparts that could be proceeded once and re-used several times. On the same way, Shared filters[28] orders the queries by containment, to limit the number of tuples for the more specific of them and avoid to check common filters several times. Nevertheless, those centralized systems depends on the server’s capacity, which may be overwhelmed when the number of users increases.

To avoid this problem, several kinds of propositions use distributed organizations to share the load on a set of collaborating servers. In classical Content Delivery Networks[35], some mirrors helps the primary sources, each of them managing a part of users’ queries. In Distributed Streams Processing Systems (DSPS), like S4[29], SODA[38] or Medusa[9], a large number of nodes work together on all the queries, having a
common query plan deployed over all the available resources. In Publish/subscribe systems with brokers, like SemCast[30], tuples are filtered and reorganized by traversing a diffusion tree, which allows to efficiently dispatch the initial streams. Those cases, if they are technically distributed, still use a centralized context, as system and users are clearly separated (Figure 1 shows the different levels of centralization in the existing solutions). But a centralized context means the users haven’t any control on their own query, the servers’ owner monopolizing all the benefits due to its exclusive and exhaustive known of the system. Oppositely, thus kind of organizations may require a high number of servers, which have a huge financial cost: it can be out of reach for a lot of publishers. Moreover, the Internet design model[11] is to avoid centralized organizations.

<table>
<thead>
<tr>
<th>Grouping</th>
<th>Resources</th>
<th>Data</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SBON[32]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SODA[38]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S4[29]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Medusa[9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FoXtrot[26]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SemCast[30]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Distributed filtering[37]</td>
</tr>
<tr>
<td></td>
<td>FlowerCDN[35]</td>
<td></td>
<td>Delta[23]</td>
</tr>
</tbody>
</table>

Figure 2: Distributed organization aspects for the concerned propositions

On the first line are the propositions in which all the users are or may be asked to work on any part of the system, even those that doesn’t interest them at all. Classically, DSPS are in this case, as all the nodes are working on a small part of a common query plan. In the applicative Multicast organizations, like SplitStream[5], each user works by her own, but all of them are involved into the diffusion of all tuples. The second line introduces a splitting of the general problem by specializing some of the nodes, regarding to semantic aspects. On the last lines comes the propositions in which users works only on the aspects that interest them. For the systems having specialized nodes, this specialization could be data-oriented or query-oriented, regarding to the desired structure.

In peer-to-peer (P2P) organizations, users take responsibility for the processing and diffusion (even if a centralized tracker could still be in charge to maintain the organization). The major benefit, contrariwise to the centralized propositions, is that the system, rather than being overwhelmed, becomes more powerfull when the number of users increase. In applicative Multicast systems like SplitStream[5] (a response to the fact that the network Multicast[15] isn’t deployed in most of the existing networks), all the users could be involved into the diffusion of all the data, each of them working on her query independently from the others. But classical peer-to-peer exchange systems, like BitTorrent[12], are more often built to take care of the user interests, by not asking them to diffuse undesired data. FlowerCDN[17], for instance, is an organization in which users provides mirrors to the data that they want, in order to obtain a sort of user-managed CDN. Semantic Overlay Networks[13] could also be used to efficiently organize the data.

In other cases, Distributed Hashtables (like Pastry[33] or Chord[34]) can be use to efficiently organize the system, as they offer a good way to dispatch informations and to explore distributed systems. Some P2P pub/sub systems like Meghdoot[19] or FoXtrot[26], for instance, uses DHT to organize the users and guide the diffusion of data. On the same way, distributed large-scale information filtering[37] proposes an extension of the Chord protocol, named DHTrie, used in their system to split the queries and decide the kind of messages that should be exchanged. Those systems are good for diffusion aspects, allowing to easily find the the subscribers that are waiting for a data tuple; but there is often redundant processing aspects.
Regarding to fact that users can also provide important processing capacities (BOINC, for instance, a distributed volunteer computing network for scientific research, provides a processing rate higher than conventional supercomputers[3]), a few DSPS like SAND[2] or SBON[32] are built in P2P context: a large query plan is deployed over all the users, that works on all the queries. In such cases, collaboration between users can be done by using distributed process sharing algorithms like MapReduce[14] that give very efficient diminution of the load, but a notable consequence is that the users are asked to work out of their interests. In particular, in a world-wide system, some users may be asked to work on aspects that are legal in the original subscribers’ country, but not in their one (for instance, a work on video streams having limited diffusion authorizations). Moreover, the organization cost of all those resources could be problematic.

In RoSeS[36], that can be deployed either over centralized or distributed systems, users having several queries contribute by combining redundant operators, which decrease the network load and simplify the processing. This is an interesting approach, but the fact that each user works independently from the others entails that there still is a lot of redundant work in the system. Such an approach generalized over all the users allows to build a collaborative system in which the users sharing common interests collaborate.

The general organization principles of distributed systems are shown in Figure 2, while Figure 3 briefly summarize the positioning of the main propositions we consider.

<table>
<thead>
<tr>
<th>Query processing</th>
<th>DSMS</th>
<th>DSPS</th>
<th>Pub/Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TelegraphCQ[7]</td>
<td>SODA[38]</td>
<td>FoXtrot[26]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4[29]</td>
<td>SemCast[30]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medusa[9]</td>
<td></td>
</tr>
<tr>
<td>Splitting</td>
<td>STREAM[4]</td>
<td>QSystem[22]</td>
<td></td>
</tr>
<tr>
<td>Rewriting</td>
<td>Shared filters[28]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Global positioning of the propositions

In grouping approaches, resources are shared to work on several queries at the same time. In splitting approaches, several parts of the queries are calculated independently from the others, the obtained results being re-used for several queries. In rewriting approaches, each query is computed separately, some of them relying on the results computed from the previous ones. Centralized systems are called DataStream Management Systems, while the others are categorized into Distributed Stream Processing Systems and Publish/Subscribe Systems.

To the best of our knowledge, the closest propositions to ours are Semantic P2P Overlays[6] and Delta[23]. In Semantic P2P Overlays, users are organized regarding to containment aspects, to limit the diffusion of irrelevant tuples: the system is built as a spanning tree, in where each node correspond to a subscription equivalent to or more specific than the parent’s one. This containment-based approach allow an efficient diminution of the useless transfers (data that are sent to the children and do not interests them, called “false positive”), which means the network load is efficiently decreased, without any losses of useful data (“false negative”), as the parent alway have at least all the data that interest the children. They do not take care of the processing aspects, focusing only on the network ones, but we can consider that their organization allows also to simplify them, as the operations performed by the parent doesn’t have to be performed again on the received data.

In Delta, subscriptions are replaced by equivalent queries using the other as views, in the aim to limit the load, and then organized to limit latency (this second operation, independent from the first one, may
re-increase the processing load, but their constraints limit this effect). The approach is so divide in several steps: first, finding all the subscription having reusable results (relevant views) and build a graph of those relations. An external algorithm\cite{16} is called on this graph to remove the cycles, as cycles at this step could be propagated to the rest of the organization. After that, they build a (limited, due to the number of views to consider) set of possible rewritings for each subscriber, then use an ILP solver to find the optimal rewritings regarding to their constraints. The latency-limitation algorithm (LOGA) is then applied to finalize the placements. The main difference, in the final organization, with the previous proposition, is that each subscriber may get his data from several sources, including the original publisher.

Both of them give so good results for decreasing and sharing the load, without asking the users to work out of their interests. The counterpart is that their organizations are hard to setup: in Delta, each subscription is compared to all the others in order to find the relevancy relation, and the next steps have to deal with graphs joining all the users. In Semantic P2P Overlays, the previously existing structure could be used to simplify this work, but there still need to compare a huge number of different subscriptions to find where to insert a new user. Even if they are distributed and efficiently limiting the processing and diffusion problems, their scalability could so be problematic due to the cost of the organization.

3 Theoretical background

As said in the previous section, our problem is related with several domains (databases’ continuous queries; pub/sub systems; DS[M|P]S...). A common part between those domains is that they can be represented by using data streams and queries over those streams. We thus presents in this section the general definitions and properties on which our proposition is based.

Data streams could take a lot of aspects, regarding to the used protocols and formats. To formally define them in the general context, we rely on the definition used in \cite{31}:

Definition 1 (Data streams) A stream is a possibly infinite set of tuples with a common schema containing two special attributes: a timestamp and a physical identifier.

Notation: we denotes $\Phi$ the set of possible streams.

For instance, an RSS Stream is a set of different items, the timestamp given in the “pubDate” tag, and the identifier by the “guid” tag, the other tags being optional. Please node that, as for mathematical sets, the order of the data tuples in the streams does not matter (as all items are timestamped and identified, any shuffle of a given RSS file describes the same stream).

The streams will be published and queried in the system by several participants. The queries expressed over those streams are determined by a query language:

Definition 2 (Query language and queries) A query language is a declarative language that allow to express queries over the data streams. A query $q$ takes a set of data streams to produce a new data stream. It may be considered as a function:

$q : 2^\Phi \mapsto \Phi$

$q(d_1, d_2, \ldots, d_n) \mapsto d_q$

Notations: $Q$ denotes the set of queries that are possible to express using the query language,

For each query $q$, we denote $in(q)$ the set of input streams used, and $out(q)$ the resulting output stream.

Several query languages may be chosen to set up a system, and we do not restrict any of them in our general model, but the organizations that are possible depends on the available mathematical relations between the queries. At least, the query language always allow to determine which queries are equivalent:

Property 1 (Query equivalence) Two queries are equivalents if and only if, applied on their regular inputs, they outputs the same stream, whatever are those regular input streams:

$$\forall q_1, q_2 \in Q, q_1 \equiv q_2 \Leftrightarrow out(q_1) = out(q_2)$$

\cite{As they are set of data tuples, the output streams are considered as equals if they contains the same tuples shown in a different order.
Some languages may also allow to rewrite the queries in order to make the computation easier. The way to rewrite a query depends on the language, but the general principle is always the same:

**Definition 3 (Query rewriting)** The rewriting operation of a query \( q \) is the searching of a query \( q' \) equivalent to \( q \) that can be preferred to provide the desired results.

For example, in SQL, the query "select distinct pid from papers;", assuming that "pid" is the primary key for the table "papers", and so having no duplicates, could be rewritten in "select pid from papers;", which exempt to compute the useless "distinct" operator. In conditional languages, the expression "\( \neg A \land \neg B \)" can be a rewriting of "\( \neg(A \lor B) \)".

In order to compute efficient rewritings, it is possible to use some views:

**Definition 4 (Views and query rewriting using views)** A materialized view is a set of precomputed data that are available for re-using. Each view is defined by a query and contain its results, allowing their use in the rewritings operation. A query \( q \) can be rewritten using a set of views \( \{v_1, v_2, \ldots, v_n\} \) when it is possible to find a query \( q' \) equivalent to \( q \), using \( \{v_1, v_2, \ldots, v_n\} \) as its inputs.

**Notation:** \( \{v_1, v_2, \ldots, v_n\} \rightarrow q \Leftrightarrow \exists q' \in Q : q' \equiv q \), in \( (q') = \{v_1, v_2, \ldots, v_n\} \).

The query answering using (materialized) views is a classical strategy ([24, 20]) to perform efficient rewritings, having its complexity depending on the chosen query language: the problem is linear for simple containment-based filtering cases, in which there’s just a need to check each view independently from the others, but is NP-Hard for complex languages like SQL or DataLog[1].

In the context of a network system, the result streams of the queries expressed by some participants could be considered as views if those participants are able to send those result streams to the others. In this case, the network cost as to be considered besides the processing cost, so the minimality of the rewritings ([10, 25]) could be an interesting strategy.

In order to know how the data are obtained, and to understand what is really computed, it is possible to reduce a rewrited query:

**Definition 5 (Reduction of a query)** Considering a query \( q \in Q \), \( \text{reduction}(q) \) is a query obtained by replacing each view in its inputs by the related query:

\[
\text{reduction}(q) = q[q_1|v_1,q_2|v_2,\ldots,q_k|v_k]
\]

where \( q[x|y] \) denotes the substitution of \( y \) by \( x \) in \( q \), and \( q_i \) is the query ran to obtain the view \( v_i \).

\( \text{reduction}^n(q) \) denotes \( n \) successive application of the function \( \text{reduction} \).

For instance, if an SQL query \( q \) is defined by “select title from V_p14;”, with “V_p14” the view defined by “select * from papers where year=2014;”, then \( \text{reduction}(q) \) will be the query “select title from (select * from papers where year=2014);”

Consequently to the reduction comes the notion of normal form:

**Property 2 (Normal form)** A query is said to be in normal form if and only if it involves no views, which mean its reduction gives the query itself, unchanged (\( \text{reduction}(q) = q \)).

The possibility to find a query in normal form by applying the \( \text{reduction} \) function a finite number of times ensures that their is no cycles that would prevent the effective computation of the final query.

### 4 Problem Definition

In this section, we propose a general definition of the organization we consider, by using the terms of publish/subscribe contexts, and then state the problem we address, after presenting how some notable existing system are categorized using our formalism.
4.1 Publish/subscribe organization

The data streams usable in the system are produced by particular nodes: the publishers:

**Definition 6 (Publisher and publication)** A publisher \( p \) publish a data stream.

**Notation:** \( \text{pub}(p) \) denotes the publication of the publisher \( p \) (\( \text{pub}(p) \in \Phi \)).

One can think that it is too restrictive to impose one and only one publication for each publisher. Except more complex notations, there is no technical difficulties to generalize to multiple publications (\( \text{pub}(p) \in 2^\Phi \)). Here, as the publication of a given publisher \( p \) (\( p \in P \)) formally noted \( \text{pub}(p) \) is unique, it can also be designated by \( p \). This abuse simplifies notations without introducing any ambiguity.

The publication of a publisher \( p \) can be characterized by a particular query \( \top(p) \) (\( \text{out}(\top(p)) = \text{pub}(p) \)). The other queries defined by the query language are issued by subscribers:

**Definition 7 (Subscriber and subscription)** A subscriber \( s \) is a participant issuing a query over data streams.

**Notation:** \( \text{sub}(s) \) denotes the subscription of the subscriber \( s \) (\( \text{sub}(s) \in Q \)).

Again, there is no difficulty to generalize to multiple subscriptions (\( \text{sub}(s) \in 2^Q \)). As done with publishers' publications, by abuse, we authorize to confuse \( s \) and its issued query \( q \) (\( s = \text{sub}(s) \)).

Having those definitions, we can fully define a pub/sub system:

(a) A basic publisher \( p \in bP \) publish an original data stream
(b) A basic subscriber \( s \in bS \) express a query without publishing the results
(c) A subscriber-publisher \( n \in SP \) express a query, and then publish the result of this query

Figure 4: The considered kinds of nodes

**Definition 8 (Publish/Subscribe system)** A Publish/Subscribe system is a couple \( (N, \text{opt}) \) where:

- \( N \) is a non empty, finite number of participants. Each participant plays at least one the two following roles: Publisher, or Subscriber. This thus determines three sets (Figure 4), which are a partition of \( N \): \( bP \), the basic publishers, and \( bS \), the basic subscribers, both of them playing exclusively the related role, and \( SP \), the subscribers-publishers, which are the participants that plays both of the roles. In this last case, the subscriber-publisher nodes intends to publish the results of its subscription:

\[ \forall n \in PS, \text{pub}(n) = \text{out}(\text{sub}(n)) \]

- \( \text{opt} \) is an optimization function which, for each subscription proposes a query which is effectively ran into the system to provide results to the subscriber:

\[ \text{opt}: bS \cup SP \rightarrow Q \]

\[ s \rightarrow q_o \]

This optimized query can use all the publishers in the system, including those who are also subscribers, to computes the results of subscriptions which are exclusively expressed over basic publishers:

\[ \forall s \in bS \cup SP, \text{in}(\text{sub}(s)) \in 2^{bP}, \text{in}(\text{opt}(s)) \in 2^{bP \cup SP} \]

Please note that our model describes only the network organization: the local optimizations does not impact the general system. We so consider that \( \text{opt}(s) = \text{sub}(s) \) as long as both queries are equivalent and having the same inputs, even if \( \text{opt}(s) \) is a distinct rewriting of \( \text{sub}(s) \).

As we associate with each node in the system a single subscription and/or publication, it is possible that some of those different logical nodes corresponds to a single physical participant:
Property 3 (Contiguity) Two nodes in the systems are contiguous when they are at the same physical location, even if their respective subscriptions are different.

Notation: \( n_1 \sim n_2 \) denotes that nodes \( n_1 \) and \( n_2 \) are contiguous.

(a) Two subscribers linked with a publisher
(b) An user subscribing several times is represented in the system by several contiguous logical nodes
(c) A device can internally perform some computations to publish different streams from several subscriptions. This is achieved using several contiguous nodes (the internal processing could be the result of local optimizations)

Figure 5: Different relations between nodes

The contiguity property may be interesting in different situations. First, two contiguous nodes can send data tuples to each other without any network operation. Second, a data stream can be send only one time to a location even if it is used by many different contiguous subscriptions use it.

For instance, brokers and mediators (see Figure 5(c)) are particular devices that inputs some data streams, possibly corresponding to several subscriptions, and may outputs several data streams to simplify the querying tasks. In our model, they would be represented by a several nodes, each contiguous, corresponding to the different network operations.

The design problem of such a system is thus to set up a reliable optimization function, regarding to the hereinabove definitions and properties, that leads to a valid organization. Intuitively, a system is valid when each subscriber gets exactly the results of its subscription. This depends on two aspects: the mathematical equivalence between the ran query and the subscription (\( \forall s \in bS \cup SP, opt(s) \equiv sub(s) \)), and the possibility to effectively find a way to obtain the needed data, without any cycles.

This second point is a reduction problem, considering that the queries in normal form are the queries that, as the original subscriptions, does not involves any subscriber-publisher node (which are the one corresponding to the previously defined views):

Property 4 (Validity) A publish/subscribe system is valid if and only if it is possible to reduce each optimized query to get a query in normal form that is equivalent to the real subscription:

\[
\forall s \in S, \exists k \in \mathbb{N} : \text{reduction}^k(\text{opt}(s)) \equiv \text{in}(\text{reduction}^k(\text{opt}(s)))) \subseteq bP
\]

4.2 Systems classification

Our formalism allow to determine several classes of systems, corresponding to different opt strategies:

4.2.1 Unicast system

An Unicast system (Figure 6(a)) is a system in which all the subscribers are directly linked to the publishers that interests them, and in which each subscription is directly ran in the system. In our formalism, this correspond to:

\[
SP = \emptyset, \quad \forall s \in bS, \text{opt}(s) = \text{sub}(s)
\]

Theorem 1 (Validity of an Unicast system) As the optimization function always associates to a subscriber it’s own subscription, an Unicast system is always valid.
4.2.2 Mirrored systems

A mirrored system (Figure 6(b)) is a system in which some mirrors nodes are used to help the publishers to diffuse their stream. Those mirrors are so subscriber-publisher nodes that get the entire stream of a given publisher (considered as unique for a single logical mirror node, regarding to our simplification hypothesis) and resend it unchanged. In other terms, a node $m \in M = SP$ is a mirror for the publisher $p \in bP$ when it express the subscription $\text{sub}(m) = \top(p)$, which is the same query that the one associated with $p$.

To optimize publish/subscribe systems, mirrored organizations allow both mirrors and subscribers to get their input data streams from publishers or from their mirrors. The optimization function is thus allowed to change the input data streams such that:

$$\forall s \in S, \exists j \in \mathbb{N} : \text{opt}(s) = s[m_1|p_1],\ldots,m_j|p_j], \text{ with } \forall k \in [1..j]:$$

- $p_k \in \text{in}(s)$, and
- $m_k \in \{m \in M, \text{sub}(m) = \top(p_k)\}$

Please note that not all the inputs are necessarily mirrored: $j$ is more formally contained in the set $[0..n]$, where $n$ is the number of publishers having at least a mirror in the system, and not always equals to $n$.

Theorem 2 (Validity of a mirrored system) As the input streams remains unchanged even if they are obtained from different nodes than their original publishers, a mirrored system is valid as long as their is no cycle between the mirrors.

Content Delivery Network[35] are such systems. They introduce mirrors, usually provided by specialized companies, and encourage, or even oblige, subscribers to get their input streams from them. The objective is to unload the publisher and to guarantee a good quality of service to the subscribers.

In the same field, Applicative Multicast systems[5] are also mirrored systems in which subscribers provide mirrors by themselves: each subscriber $s$ provides one mirror $m_i$ for each of his input streams $i \in \text{in}(s)$, with, $\forall m \in \text{in}(\text{opt}(s)), m \sim s$.

4.2.3 Rewriting systems

A rewriting system (Figure 6(c)) does not introduce other nodes than the basic publishers and the subscribers, but may ask to each of the subscribers to send its results to the others. It rely on the concept of answering queries using views, by considering each existing subscription as a possible view.

In our formalism, this means a subscriber also act as a publisher, publishing in the system the result of the query he computes. This intuition can be formally captured with a simple condition:

$$N = bP \cup SP$$

This gives much more possibilities to the optimization function, regarding to the rewritability relations given by the query language; but consequently, those kind of system may be harder to set up in a valid way.
As in mirrored system, avoid cycles is an essential condition; but the organization must also ensure that the used rewritings does not cause losses of data:

**Theorem 3 (Validity of a rewriting system)** A rewriting system is valid when it rely on valid rewritings rules (having the result of the function opt equivalent to the subscription for each subscriber) and when their is no subscriber recursively depending on itself (no cycles).

Delta[23] and Semantic P2P Overlays [6] are both instances of (valid) rewriting systems, as will be QTor. The second of those examples is also containment-based, which mean each rewriting use only one view: this is much simpler to setup, but may gives more limited results.

### 4.3 Problem statement

Besides the validity, the opt function has to take care of several aspects to be reliable. In other terms, as this is an optimization function, the organization choices have to determine what are exactly optimized.

#### 4.3.1 Evaluation criteria

Classically, pub/sub systems consider three main aspects: the latency, the processing load and the diffusion load, partially depending from each others. Besides those functional aspects, the organization overhead has also to be studied, as the possibility to deploy a system highly depends on it. It is not possible to optimize all of those aspects, due to the fact that they are partially antagonistic, so each kind of proposition has its own priority between them.

**Definition 9 (Diffusion load)** The diffusion load represents, for each publisher, the load required to send the published data to all of the subscribers directly linked with it. It so depends on the number of those subscribers, and on the volumetry (number and size) of the data.

Decrease the diffusion load of the basic publishers is the main objective of mirrored systems. The mirrors are so used to share this load between several nodes, each of them having less children to deserve. But the global diffusion load remains high, as each data tuple still has to be sent to each subscriber.

Some other kinds of systems, such as Semantic P2P Overlays, aims to reduce the data volumetry, which helps also for the processing load:

**Definition 10 (Processing load)** The processing load represents, for each subscriber, the load required to obtain the subscription results from the received data. It depends on the volumetry of those data, and on the complexity of the query that has to be applied on it.

Having the subscribers correctly linked with the others, each of them publishing only the results of its subscriptions, is a good way to reduce the processing load, as a lot of irrelevant tuples (“false positive”) can be dropped by the previous subscribers without having to be studied.

Moreover, the rewrited queries are also often simpler than the original subscriptions, allowing to win on both aspects of the processing.

**Definition 11 (Latency)** The latency represents, for each subscriber, the delay between the publication of a new data tuple by a related basic publisher, and the obtaining of the updated subscription results. It depends on the network state (number of connexions and transition time for each of them) between publisher and subscriber, and on the availability of each intermediate node.

In other terms, as the high number of operations to perform leads to postpone the tuples sending, overwhelmed nodes slow the systems even if the network state is good. Delta has shown[23] that the slowing due to an excessive diffusion load could be worse than the one due to a reasonable distance between nodes.

It is thus required to find a compromise between the load aspects, that leads to highly separate the last subscribers from the basic publishers, and the network aspects, that leads to each node to perform a lot of operations. In Delta, for instance, the first part of the organization works to limit the processing load, and then the LOGA reduces the global latency by partially re-increasing this load.
Definition 12 (Organization overhead) The organization overhead represents the work needed to keep the system reliable despite the dynamic aspects (churn, volumetry changes...). It depends, of course, on the number of connections that has to be broken and rebuilt (the network stability), but also on the number of operations that has to be performed to know how the network should be changed (the cost of the function opt itself).

Indeed, the insertion or leaving of a subscriber, for instance, could leads to perform a high number of operations (like compare the related subscription with a huge neighborhood), to finally find out that only a few connections have to be impacted. On the other hand, a simple operation like the insertion of a new mirror (that doesn’t need any subscriptions comparison) could impact a lot of connections.

As a non-functional aspect (the work done for organizing the system does not directly impact the query processing nor the data diffusion of each node), this setup part could be seen as secondary; but it is important to note that it may cause major scalability problems. The deployment of a huge system highly depends on the ability to efficiently perform those setup operations.

4.3.2 Our objectives

With QTor, our aim is to limit this organization overhead, while having an efficient decrease of the processing load: our algorithms have to find, for each subscriber \( s \), the \( \text{opt}(s) \) having a cost as minimal as possible, with an incremental approach that exempt to evaluate as much as possible of the already inserted nodes, and that limit the impacts of the relinkings, to guarantee the scalability.

In other terms, we do not rely on a global optimization, that could give better processing results, but need a huge cost to setup, as we expect that the gain for processing aspects does not justify the required organization cost. As a counterpart, our organization is so built to be always optimized, without having to use strategies such as “this optimization algorithm runs every \( X \) insertions”, which ensure that the \( X - 1 \) previous subscribers won’t be optimized during a long time.

Focusing on the processing load and the organization overhead, we so consider the limitation of latency and diffusion load are secondary: our organization algorithms reduce the latency as often as possible, but without allowing this reduction to re-increase the processing load (unlike Delta’s LOGA). About the diffusion load, we consider the fan-out of each node (the number of children linked with a given parent) as a major constraint, but prioritize the latency over it as much as possible (which mean the fan-out, even if it is never exceeded, is reached any time the number of descendants is equal or higher).

5 The QTor approach: collaborating communities

5.1 Communities: an abstraction to head QTor

As users in a pub/sub system may express similar queries, we propose to group them into communities. Those communities are an abstraction, and focusing on them rather than on the users allows to reduce the complexity of the problem on all the organization aspects.

Definition 13 (Communities) Considering a pub/sub system \( \langle N, \text{opt} \rangle \), a community is a group of subscribers sharing similar interests, which is materialized by the fact they issue similar subscriptions. The community of the subscriber \( n \) is noted \( c(n) \).

\[ \forall n, n' \in SP, c(n) = c(n') \iff \text{sub}(n) \equiv \text{sub}(n') \]

Communities are thus equivalence classes on the set of subscriptions. One of those equivalent subscriptions, for example the first’s user one, normalized if allowed by the language, is noted \( \text{sub}(c) \) and used to label the community.

\[ \text{There is no problem with using and incomplete equivalence relation to build the system: if it is not possible to find out that some queries are equivalent, the system would just create more communities that could be needed, but the impact on the final organization will be limited.} \]
We note here \( C \) the set of the communities in the system, which is a partition of \( N \) (considering the publishers can have their own communities, grouping their mirrors if they have ones).

Those communities allow to share the load (data acquisition, query processing and diffusion) between different subscribers, and to ensure that every subscribers in each community receive the same data. They also are \textit{independents}, which mean they may have different inner organizations, and that changes on those organizations does not directly impact the other communities.

\textbf{Property 5 (Validity of a system using communities)} As all the subscribers in a community are waiting for the same results, a pub/sub system using communities is valid iff:

- each community work on a query in normal form that is equivalent to its labelled subscription:
  \[
  \forall c \in C, \exists k \in \mathbb{N} : \text{reduction}^k(\text{opt}(c)) \equiv \text{sub}(c), \text{in}((\text{reduction}^k(\text{opt}(c)))) \subseteq bP
  \]
- each user in the community receive the results obtained by the computation of this query

\subsection*{5.2 Organizing communities using a query hypergraph}

According to the rewriting principle described above, we consider that every community can be used as a view to rewrite other communities queries. The system can thus be described as a graph representing the relations between the different communities. On Figure 7, we can see the successive simplifications given by this model: using communities helps to decrease the number of considered relations, and then it become possible to choose the most interesting of them, in order to finally link the subscribers. The top-graphs are the abstractions given by the \textit{QTor} approach:

\textbf{Definition 14 (Query hypergraph)} Considering a pub/sub system with communities, we define the query hypergraph as the graph of all the possibilities to rewrite each query: a set of communities \( \{c_1, \ldots, c_i\} \) is linked to a single community \( c \) if and only if it is possible to rewrite \( \text{sub}(c) \) using the set \( \{\text{sub}(c_1), \ldots, \text{sub}(c_i)\} \) as views. More formally, the query hypergraph is the 2-tuple \( (C, \rightarrow) \) with:

- \( C \) the set of communities
- \( \forall c_0, c_1, \ldots, c_i \in C, \{c_1, \ldots, c_n\} \rightarrow c_0 \leftrightarrow \{\text{sub}(c_1), \ldots, \text{sub}(c_i)\} \leadsto \text{sub}(c_0) \)
This query hypergraph is shown at the top-left of the figure. It is much simpler than the same hypergraph directly over the subscribers rather than the communities, shown at the bottom-left, and allow to highly limit the number of relations to compare between queries: the insertion of a new subscribers depends on the number of distinct communities in the system, and not on the number of nodes.

But those relations are still theoretical: not all of them will be exploited in the resulting system. The hypergraph shown at the top-right represents the relations that are chosen to be use:

**Definition 15 (Community hypergraph)** Considering a pub/sub system with communities, we define the **community hypergraph** as the hypergraph of all the effective relations that are used in the system: a set of communities \( k \) is linked to a community \( c \) when the subscription of \( c \) is (and not only can be) rewrited using all the views given by \( k \). More formally, the community hypergraph is the 2-tuple \( \langle C, \to \rangle \) with:

- \( C \) the set of communities
- \( \forall c_0, c_1, \ldots, c_i \in C, \{c_1, \ldots, c_n\} \to c_0 \iff \text{in}(\text{opt}(c)) = \{\text{sub}(c_1), \ldots, \text{sub}(c_i)\} \)

As all the links are build regarding to the rewriting aspects, the equivalences between query and subscription is ensured for each community, which is the first condition for having a valid system, as long as there is no cycles. Using this hypergraph, this is simpler to go down to the users, to link them into a collaborating graph shown at the bottom-right and representing all the sending relations that are effectively ran in the system.

The system organization is thus split into three parts:

- grouping the users having equivalent subscriptions into communities
- connecting the communities to each other regarding to the rewritability relations
- applying those relations onto the subscribers

### 5.3 An incremental way to organize the system

The second important aspect of the QT or approach is that the organization is fully dynamic and incremental. A lot of distributed system, including Delta, needs to be periodically reevaluated from scratch, which mean the system recalculate the desired position for all the nodes without taking care of the existing links between them. We consider that this implies an organization much harder than needed. In our model, the advantages of using communities would be highly decreased if the only way to group the subscribers was to compare all the subscriptions in the system from scratch.

To avoid this problem, the best way is to use the previous organization rather than to reset it. As the mathematical equivalence between two subscriptions will never change while both of them are present in the system, we never need to dispatch a community. Any inserted subscription has thus to be compared with each community, which represent, as said in the previous subsection, a much lower work than to compare it with all other subscriptions.

Moreover, in an incremental organization, the query hypergraph itself has never to be fully calculated: as the community hypergraph is always available, because he represent the current state of the system, it can be used to build the part of the query hypergraph that is needed for each operation: starting from the publishers, an exploration following the existing links will give us all the views in the system (there is no unlinked community).

As each community will be inserted (or moved) independently from the others, the relations between the considered community and the others is the only part of the query graph that is interesting when it is used. We can so prune the previous search, in order not to consider the irrelevant views: if a given view cannot send any part of the needed tuples, none of its exclusive children, at any depth, will be able to rebuild those tuples, and so none of them will be relevant. Non-exclusive children may be relevant, but this implies that they have other parents that are relevant (and so not pruned), so the pruning won’t cause their loss.
If the communities are able to know their direct parents and children in the data diffusion, they so provide an efficient way of searching placements, allowing the system to be perpetually (rather than periodically) optimized, with, once again, a complexity much lower than the exploration of all the system.

5.4 Subscribers’ collaboration inside a community

We saw that communities allow to simplify the problem at macroscopic level (complexity of insertions), by reducing the number of nodes to consider, and by managing neighborhood links. But they also help at microscopic level (number of nodes to affect): as they are independents, the links between users inside each of them may be reorganized without impacting the other communities.

Semantic P2P Overlays [6] use such a characteristic of equivalent relations at the leaving of a subscriber: if there still is in the system a subscription equivalent to the one that disappeared, they only have to ask the corresponding node to take the vacant place, and the system remains valid, without having to move all the children. QTor generalizes this characteristic to all the subscriber-level events: creating or removing a community will obviously impact the community hypergraph, but all other insertion or deletion of subscriber will only need to impact its community (the parent and children communities may choose to reorganize themselves in order to improve the diffusion, but this is an optional operation).

Moreover, the subscribers inside a community can collaborate together. In Semantic P2P Overlays, a single subscriber is considered as strong enough to acquire the data, work on them and resend them to the others, which can accept the data without any work. In QTor, we consider that it is possible for some subscribers to collaboratively work on those different aspects (data acquisition, query processing and results diffusion) in order to share the load between them.

As each community is independent, they may have different inner organization, regarding to the work that must be done (number of providers, different kind of operations to perform...) and/or to the preferences of the users. As all the subscribers inside a community have the same interests, there is no problem with deploying a cluster or grid organization at this level, if it could be useful. We do not constrain any specific way to organize the community, as long as all the subscribers inside them and all the children communities receives the needed results.

6 Application

In order to apply the promising aspects of the general QTor approach, we setup an example of organization workflow, shown on Figure 8 and triggering the two types of events: insertion of a new subscriber and the leaving of an already inserted subscriber. The moving of a community can also be triggered, but, for now, as the consequence of one of those events. We explain in this section what choices have be done to obtain our system model.

6.1 Strategic choices
6.1.1 Community inner organization

In order to evaluate the basic advantages of the QTor organization in simple cases, we choose not to deploy any collaborative work solution for now: the strongest subscriber in the community is in charge to get the data from the previous communities, to compute the given query and to resend them to his nearest neighbors. This is also the strategy chosen by the foregoing two papers (in both cases, if a subscriber can receive directly his results without any work, it will do).

In order to have any subscriber in the community receiving the results, we so deploy a spanning tree, ordering the nodes regarding to their weights (acquirers from crowded communities may has to be prioritized, as a lot of subscribers may depend of them) and sending capacities, to limit the general latency.

In future works, communities may compute statistics on its inputs/outputs (volume of acquired data and produced results), in order to decide if it has to try to move when new opportunities appears, and to
6.1.2 Relation between parent and children communities

In order to avoid problems of overwhelmed providers, any community \( c \) having at least one other community as parent (\( i.e. \) any community that does not involve a basic publisher) has to send some resources to the parent communities. This is done by asking some of the users having subscriptions in \( c \) to subscribe to the considered communities too. Those new subscriber nodes has no need to be inserted following the complete procedure: as their equivalent communities are already known, they can be inserted directly in them.

A link between two communities \( c_1 \) and \( c_2 \in C \) in the community graph is so implemented by the existence of two nodes \( n_1 \) and \( n_2 \in S \) with \( n_1 \in c_1, n_2 \in c_2, n_1 \sim n_2 \).

Those subscribers helps the parent community as much as they want, giving it at least the needed resources to send the results to one of its other users (in order to replace the sending they consume to get those results). As they are physically equivalent to one of the child community nodes, they also work as acquirers for this children community, providing its needed data.

An interesting consequence of this load-sharing strategy is that a community will never disappear if it has children: even if all the real subscribers lefts the system, the acquirers sent by the children communities can still maintain the community alive. Even if this leads to move the children community (as they becomes in charge to compute the data from their parents too, which could give a processing load higher than if they where placed somewhere else), there so is no chance to have to move them in a hurry, as the data will still income while the new place is searched.

6.1.3 Insertion of a new subscriber

As seen in the previous section, the insertion of a subscriber may have limited effects if it has only to join an existing community: the considered community reorders without any changes on the rest of the system. In the case that no existing community has a query equivalent to his subscription, a new one must be created,
and then placed in the system. The complete workflow may so be applied in such cases: the searching of equivalence (which give no results), and then the searching of the best rewriting and the placement of the community.

The inner organization of a community containing only one user is trivial, but the chosen providers may have here to reorder, as they receive a new subscriber sent as acquirer by the newly created community. Moreover, the creation of a new community creates some opportunities that the other communities may wants to use in order to decrease their own load: a signal is so sent over the system to inform that the new community was created.

6.1.4 Leaving of a subscriber

When a subscriber leaves the system, its community may have to be reorganized, but there’s no notable problem if the community still gather some other subscribers. As the children communities send acquirers to each of their parents, there will not be any case in which a community completely disappear while having some children, so the community reorganization algorithm is the only work to do there.

On the other hand, if an acquirer from a parent community becomes the only member of a community, his original community may decide to move to more interesting parents, but this is not a systematic case, and can be considered as similar as all other community moves.

6.1.5 Moving of an existing community

As said before, community movings could be, in a complete system, triggered by several kinds of events (data volumetry changes, too much charge in the parent communities...), but we considered for now only the case in which a new community appears in the system, in order to have an efficient logical placement with a community hypergraph remaining quite stable. To keep the consistence of this hypergraph, several community moves cannot be done at the same time, so we consider, for now, the presence of a mutex system, for example implemented by communicating with a tracker, authorizing only one move-checking at the same time.

There could be several ways to perform such operations, regarding to the available informations for each community: if they don’t have any information about the existing hypergraph, the communities that try to move could have to follow the complete procedure, as for the insertion of a new one: searching relevant views, then building all the available rewritings, and then check if one of them is better than the current. But we can consider that those steps were performed at least once, when the community was created: each community could so cache those informations, in order to only have to update them when they receive a signal informing that the graph where changed (signal containing the identity of the inserted or removed community). Having this mechanism allow the search of a new placement quite simple: there’s only to work, at the reception of a signal, on the rewritings involving the newly inserted view.

An important point to note is that, in an incremental approach such as ours, moving an existing community is the only operation that could create cycles in the community hypergraph (as the insertion of new communities make those new communities being leafs). But those cycles could be easily avoided: the communities that try to move have only to check if the possible parents are some of their children or not. This could be done by a cache mechanism (that require to also signal the movings to the parent communities), or by an exploration, before starting the search, of all the children, to mark them as irrelevant. In both cases, this operation as so a limited cost, counterbalanced by the fact that those children community will so not being used in the search, which simplify it. Rather than having to call an additional algorithm to remove the cycles, our organization so allow to prevent their formation by having a cost being at worst the same as if cycles weren’t studied.
6.2 Algorithms

6.2.1 Searching equivalences

When a new subscriber joins the system, the first step is to explore the existing community graph to search if there’s already a subscription equivalent to his one, as there is no need to do anything but inserting him in it in this case. As said before, this step can rely on the former organization to be more efficient, searching on the existing community graph rather than on an unlinked set of subscribers or communities.

Algorithm 1 (Searching an equivalence in the community hypergraph)
inputs: new subscription \( s \), community hypergraph \( G \)
outputs: the found equivalent community, if any.
let \( C \) the stack of nodes to explore.
insert into \( C \) all the original publishers of \( in(s) \)
while \( C \) is not empty, do
    pop the first element of \( C \) as \( c \)
    if \( c \) is not marked, then
        mark \( c \)
        if \( s(c) \equiv n \), then
            return \( c \)
        else if \( s(c) \) is relevant to \( n \), then
            add all children of \( c \) into \( C \)
    endif
endif
endwhile
return not found

This algorithm runs in \( O(\text{number of communities}) \), with a lot of pruning opportunities regarding to the relations. We consider here that the subscription of a community is \textit{relevant} when it can send a part of the data needed by \( n \), as an equivalent community that is not linked to the publisher is necessarily connected to such a provider. Pruning irrelevant communities’ children has no chance to be problematic, as the relations are transitive: if there is relevant children deeper in the graph, there is at least one other way to find them.

6.2.2 Selecting the best rewriting

Selecting the views. Entering this step in case of subscriber insertion means that there was no equivalent community found. A new one is so created, and we start searching the placement for it.

Algorithm 2 (Searching relevant views on the community hypergraph)
inputs: new community \( com \), community hypergraph \( G \)
outputs: the list of possible rewritings (containing at least the subscription itself)
let \( V \) the list of relevant views.
let \( C \) the stack of nodes to explore.
insert into \( C \) all the original publishers of \( in(s) \)
while \( C \) is not empty, do
    pop the first element of \( C \) as \( c \)
    if \( c \) is not marked, then
        mark \( c \)
        if \( s(c) \) is relevant to \( n \), then
            add all children of \( c \) into \( C \)
            add \( c \) into \( V \)
    endif
endif
endwhile
let $R$ the list of available rewritings
add $\text{sub}(\text{com})$ into $R$
fill $R$ of all possible rewritings of $\text{sub}(\text{com})$ using $V$
return $R$

The result $R$ of this algorithm so corresponds to all the links in the query hypergraph that have $\text{com}$ as destination, without any other part of the hypergraph, which allow to search the best place for the single community $\text{com}$ without having to deal with all the others.

Possible optimization: the search part of this algorithm is very similar than the previous one, following the same paths in the graph, and both can in fact be applied at the same time for optimization reasons. Some other heuristics than the relevancy may also be used to prune some part of the hypergraph, but we consider for now that it is better to fully explore the relevant branches.

Building rewritings. The filling of $R$, in the previous algorithm, cannot be described in the general case, as this operation highly depend on the chosen query language. We consider that having such an algorithm is a precondition to deploy $QTor$. For instance, here is the filling algorithm used for our simulations, in a case of filtering language, in which each query will consist in a set of filters:

Algorithm 3 (Filling the rewriting list for filtering)
inputs: subscription to rewrite $s$, list to fill $R$, set of relevant views $V$
for each view $v$ of $V$, do
  if another view $v'$ in $V$ contains all the filters of $v$, then
    remove $v$ from $V$
  endif
endfor
for each subset $V'$ of $V$, do
  if it is possible to rewrite $s$ using $V'$, then
    add the rewriting $q$ into $R$
  endif
endfor

Possible optimization: It is possible to skip some rewritings if it is possible to know that they won't be used. For instance, as the subscribers may has to send resources to the parents' communities, a rewriting using too much views may be more difficult to use than the original subscription. We so can stop the recursive calls when $|\text{in}(q)| > F_{\text{com}}$.

Selecting placement. Once the list of available rewritings is known, the next step is to select the one that will be the most interesting at the moment.

As seen in Section 4, we use here the resulting complexity as the main choice argument, considering the simplest the rewrited query is, the better. Some others informations can be used if they are available, for example the average volume of data received, as we will explore in a future work. In some case, it should be theoretically possible to choose a rewriting that give not the minimal complexity, but is better for some other aspects (e.g. latency), or that use other load informations than query complexity (e.g. data volumetry), but those cases either aren’t considered here.

We so rely on a cost model, with a cost function that associate with each query a score, considering the lower this score is, the better. For now, this cost function only call the $x$ function used in Section 4, but it can be easily redefined to improve the placement:

Algorithm 4 (Selecting the placement of a community)
inputs: rewriting list $R$
let chosen the chosen rewriting, initially $R[0]$
for $i$ from 1 to $|R| - 1$, do
  if $\text{cost}(R[i]) < \text{cost}(\text{chosen})$, then
chosen ← $R[i]$

endif
endfor
return chosen

This algorithm runs in $O(|R|)$, which depends on the number of views selected by the graph searching, and return the query that will be used as $opt\text{(com)}$ until new opportunities appeared. After linkings, a signal is emitted to the other communities, to inform them that there is a new placement opportunity. Regarding their situation, they can choose to stay where they are, or to try to move.

Possible optimization: in a containment-based context like in Semantic P2P Overlays, only one view can be used to rewrite a query. In this case, there is no need to collect all the views, then apply a filling algorithm, and then launch a distinct algorithm to select the best available rewritings: all those steps can be gathered in one, by trying to rewrite and use the cost model during the hypergraph search algorithm. This allow, for simple cases in which there is no need to rely on several parents for a single query, to compute the organization with a single loop.

6.2.3 Inner organization

As for the macroscopic organization, the community inner organization must be stable: as long as there is no need to break the existing relations between nodes, it must remain unchanged. But, as we aim to limit the latency, it is important to reevaluate the situation at each event, rather than periodically.

Considering the community as a tree, we so compare the effective and theoretical average node depth at each insertion or leaving: while the tree obtained by simply linking the disconnected nodes at the first available position has the same shape than the ideal tree, we use this solution, no matter if some nodes are misplaced. Once both tree are different, the algorithm moves the nodes to their best place by keeping as much as possible of the existing links.

The building of the ideal tree is a simple task: it consist in ordering the nodes regarding to their weights and capacities (the heaviest or strongest at the head), and then split the list into several floors, each floors containing as much as possible nodes regarding to the sum of the capacities from the previous (no matter which parent each node exactly have in the previous floor). Note that if the community contains a basic publisher, this node has always to be at the head, even if some other nodes are heavier or stronger.

Considering a function $depth_T(n)$ that indicates the depth of a node $n$ in the tree $T$, the reorganization algorithm is:

Algorithm 5 (Community inner reorganization)

inputs: effective organization tree $E$, ideal organization tree $I$

for each node $n$ in the community, do
  if $depth_E(n) \neq depth_I(n)$ then
    unlink $n$ from its parent
  endif
endfor

for each node $n$ in the community, do
  if $n$ isn’t correctly linked then
    link $n$ to the first available position with a parent $p$ having $depth_I(p) = depth_I(n) - 1$
  endif
endfor

This thus ensure that the deepness of the effective tee is always the smallest, with the minimal possible linkings (if disconnecting a node from its current parent has no change to decrease the depth, this operation is not performed).
7 Evaluation

As our aim is to evaluate the benefits and the scalability of our approach, we choose an application in which our different referents can be deployed. It consist in a filtering language, allowing logical several kinds of operations. We show in this section the setup we deployed and the results we got on the different aspects.

7.1 Experimental setup

7.1.1 Query language

In order to be compatible with Semantic P2P Overlays, we choose a filtering language with the following characteristic:

- Data remains unchanged after query evaluating. Queries filter, and do not transform.
- Filters are conjunctives (boolean operations, using the classical logical and operator)
- Filters can consists in keywords that must be in the text or in the metadata

This query language so gives the properties required in Section 3: the query equivalence is easily checked by the fact that two equivalent queries consists in the same set of filters. As in Semantic P2P Overlays, the rewriting possibilities are based on the fact that the keywords already filtered by the parents does not have to be filtered again, allowing the children to perform only the part of their keywords that were not previously filtered. The problem of rewriting queries using views in such a language is so linear, having to compare the query to rewrite to each view independently. The reducing of a query simply consists in checking the keywords filtered by the parents.

7.1.2 Baseline organizations

For comparison, the two classical controls were ran: a Unicast system, in which the publisher send all the data directly to all the subscribers, which run their subscriptions as (optimized) query; and an applicative Multicast system, in which the users sends all the data to each other, and then run their subscriptions without sharing their results (the placement of mirrors nodes given by the subscribers using our inner-community algorithm). We also ran a third control experiment: a collaborative system, in which users are grouped into communities, but without organizing those communities with each others, in order to evaluate the contributions of the different parts of our approach.

On the other hand, we compare our organization with a modified version of Semantic P2P Overlays: as the original proposition didn’t take care of the fan-out constraints, considering each user as able to send all the data to all the others in the system, we added an extension algorithm to introduce this more realistic aspect. The concept was, in case of equivalence found, to replace in the diffusion tree the first node being less powerful or working of a child query by the newly inserted node, and then to get down as much as possible of the children nodes, in order to limit the distance.

We still are working on the use of multiples views to build the rewriting, and on the comparison with Delta, so the related results aren’t shown in this paper.

7.1.3 Experimental conditions

We ran our experimentations over a synthetic case, obtained with randomly chosen keywords from both data and subscriptions, with those characteristics:

- the number of keywords in subscriptions follows a Poisson distribution, $\lambda$ 5.
- the chosen keywords in each data/subscriptions follow a Zipf’s law, with skew 4.
- the fan-out for each node follows a Poisson distribution, $\lambda$ 30.
100 sources were considered, each of them having a number of subscribers growing to 20,000, and the shown results are the average of those different cases. Data tuples produced by each source followed the same characteristics than the subscriptions. Each source published 500 tuples for each step of experimentation. That could looks like a limited number of tuples, but remember that our proposition is query-oriented: the published tuples has no impact on the organization, and the only thing we has to know from them is the proportion of gain on the loads.

The experimentations were run on a virtual machine, with a simulator writed in Java using PeerSim[27], and reproducing a realistic situation: a tracker node was in charge to indicates to each new subscriber its community position, and then the community had to reorganize itself (for controls having no communities, the tracker was in charge to fully place each of the nodes).

7.1.4 Future experimentations

Besides the Delta comparison that we are working on, we plained to set up several kinds of experimentations, in order to fully evaluate our system.

First, a prototype is currently writed by a part of our research team, using the same engine as the algorithm, but allowing to deploy the organization to a really distributed system.

A set of data tuples/subscriptions is also currently studied, based on real data. It use a set of RSS items collected by the CNAM[21]. To obtain relevant subscriptions on those data, we used the Standford’s Named Entity Recognizer[18], and we plain to compute the popularity of each subscription from the number of emitted data containing each keyword, considering that publisher and subscribers will often have the same hot topics.

We also plain to explore organizations given by several other languages, for instance a filtering language allowing all the classical boolean operations (logical and, or & not).

7.2 Experimental results

7.2.1 Organization overhead

Figure 9(a) shows the number of views analyzed during the initial equivalence search: in the QTor approach, the use of communities allow to highly decrease it, using at worst one check per community rather than one check per subscriber. The real QTor results are also better than the “non-cascaded QTor ” ones, because of the pruning opportunities given by the cascade.

Figure 9(b) shows the number of views that has been informed that a new view was created (this so happen only when a subscriber join the system without any previous node working on the same subscription). On each case, there where about ten nodes that where able to use the newly inserted view, and at most four that were effectively moved. Once again, the high difference between Semantic P2P Overlays and QTor comes from the use of communities, that highly decrease the number of nodes to consider.

7.2.2 Query processing

Figure 9(c) shows the number of data received by the users. Unicast and Multicast reach of course the maximum: in those systems, each user has to receive all the published data. Even if the QTor using the full algorithm creates some duplicate data (the relevant tuples are those coming from all the parents), the difference is not visible, as those duplicates comes only at the head of some of the communities.

For the other cascaded models, no duplicates where created; on both cases, the organization implies that the unwanted tuples are not transfered to the children, causing a decrease of the tuples to evaluate. A part of this decrease comes from the equivalence between the queries (shown by the “non-cascaded QTor ” results); the other part is given by the cascade.

Figure 9(d) shows the number of filters that where effectively been analyzed to evaluate all the received data tuples. The high decrease is due to the limitation of incoming tuples, but also to the limitation of redundant checks: in the “non-cascaded QTor ”, the subscribers at the head of their communities applies
Figure 9: Experimental results
their entire subscriptions, but the others does not have to do any processing. Moreover, in the cascaded models; those subscribers has a query that contains less filters than their original subscriptions.

Semantic P2P Overlays has a little worser results, due to the fan-out reasons: it is not always possible to move the desired nodes, as the target parents may reach their fan-out before that the considered nodes could try to contact them, so some optimization opportunities are lost (in our QTor setup, this cannot appears, as the children communities sends resources to the parent ones, always preventing them from being overwhelmed). Remember also that, for more complex operations than simple keyword filters, QTor allows users in the same community to collaborate, which would be much more difficult in Semantic P2P Overlays. So, the decrease due to community organization may means that the work is shared by several users, rather than done by only one of them.

Note that the network & diffusion aspects are also shown in Figure 9(c), as the reception and emission of tuples are symmetric.

7.2.3 Latency

Figure 9(e) shows the average distance from a subscriber to the publisher, which is the counterpart of the query processing. In Unicast, all the subscribers are linked to the publisher, which mean that a published having a fan-out about 30 like the other nodes is always overwhelmed), so the distance is always 1. In Multicast, the mirrors provided by the users are efficiently organized into a community, most of those nodes (including the publisher) reaching their fan-out, which leads to the second smallest distance.

For the three other organizations, the distance is partially due to the logical aspects: in the “non-cascaded QTor”, a multicast-like organization is built separately for each community. In the cascaded ones, the aim to limit as much as possible the processing load leads to place the subscribers as far as possible from the publisher (Figure 9(f) shows the deepness in the logical tree). QTor gives here better results than Semantic P2P Overlays due to the fact that the sending of capacities to the parent communities allows to deserve some children users without reaching a fan-out, and that communities helps to efficiently place the nodes regarding to weights aspects, that becomes more important while the size of the community graph increases.

8 Future works

8.1 Distributed organization and dynamic aspects

For simplicity reasons, we considered for now having a tracker that calculate the ideal position of each node in the graph; but we plain to setup a fully distributed system, in which the nodes can join the system by themselves. This is made possible bu the fact that out organization is open and quite simple.

Having each community able to inform which are its children, all the community hypergraph exploration could be done by the joining node itself: it only has to contact the publishers’ communities to start its exploration, and then to contact each relevant community using the same algorithm than the tracker (this operation would be longer, due to the network transfers to discover the network structure, but this do not cause any major problem) If there’s no equivalent community, the node will have to calculate the rewritings itself, and then to inform the system it just created a new community (which could be done by simply send a message to the publishers communities and let them propagate it).

The insertion of a node in an existing community will be a more classical peer-to-peer problem. Without any tracker, each community has to organize itself, regarding to the available resources and the (unique) query to compute. Some members of our research team are currently working on those aspects.

An interesting aspect of this configuration would be that each community could organize itself differently to the others, but could also choose different metrics to decide where they place themselves. For instance, a community having strong resources could choose to go nearer the publishers, sacrificing the processing load diminution to have a better latency.

A fully distributed system could also give some metrics that are difficult to evaluate for a centralized tracker. For instance, a node in the community could gather some statistics about the data volumetry, in
order to inform the other communities: they so could decide if they should move regarding to the current activities of the publishers.

8.2 Reliability, security and access control

Of course, an important aspect to evaluate, in case of distributed systems, is the reliability of the data exchanges. In a simulated case, there is no problem with having only one route between the publishers and each subscribers; but in real applications, as the nodes could disappear without preventing, this is important to have several ways to get the desired data.

The QT or approach allows to setup such configurations easier than in the other rewritings systems: the existence of communities encourages the users having equivalent subscriptions to communicate with each others, and so to setup, if required, several diffusion tree to prevent the churn (this is one of the currently studied aspects of the inner organization). Moreover, the mechanisms of sending resources to the parent communities makes the macroscopic relations stronger, as it prevent the communities to disappear suddenly. If needed, it can be extended to several acquirers rather than only one.

Another interesting aspect is that the joining of an existing community could depend on some constraints, for instance the verifying of access authorizations to the considered data: if a node express a subscription it has no right to execute, the tracker, or the community itself in case of fully-distributed deployment, could answer that it is not allowed to join and get the results, and maybe suggested to join another community corresponding to the data it can get.

The security aspects, for instance the response to a malicious node attacks, still has to be studied, but we expect that the community organization could help, allowing to identify easily the nodes that can collaborate and the desired results.

8.3 More uses of the contiguity

Our simulation did not allowed for now a single participant to express different subscriptions. As said in the previous sections, different subscriptions expressed by the same participants (which are represented by several contiguous nodes) does not impact the macroscopic structure, which was our main objective in this paper.

But the existence of several contiguous nodes having related subscriptions could have some effects in the concrete organization: those nodes placed at the head of the community, proceeding to the evaluation of their queries, allow the physical participant to use some local optimizations that could decrease the processing load. Moreover, if one of the subscriptions of such a node is rewrote using one of the others, and if this nodes computes each of the queries, the global latency is decreased by the fact that the children community gets its results without having to wait the propagation in the parent community.

Sending enough resources to the parents communities to be in charge of their processing could so be strategy for communities that want to prioritize latency over processing without modifying the processing-oriented cascade, and so without having concurrent organization rules.

As a consequence, even if we encourage distributed solutions, we can point out that the QT or approach is compatible with centralized ones: a powerful participant can subscribe to each community, get in charge to proceed the queries, and then apply any efficient local optimization to compute the results for all the other participants, as in DSMS (which can reduce the global latency to the results obtained by the “non-cascaded QT or ” simulations). The main difference between a centralized system and such a deployment is that the organization remains open, allowing the nodes that do not want to rely on the unique intermediate to obtain their data differently.

8.4 Source discovering and one-shot queries

Considering an application like, for instance, the following of several blogs in the Internet context, we can expect that the users would like to received all the data relevant to their interests even if their publishers where unknown for their original subscriptions and then not appear on them. Our proposition allows to
easily take care of such cases: as much of the existing communities do not communicate directly to the
original publishers, there is only a few nodes that have to be linked with any newly incoming publisher.

If the publisher describe the data tuples it produce by a query a little more specific than just $\top(p)$ (for
instance, by giving a list of topics or other informations about the tuples structure), the rewriting algorithms
could insert it in the hypergraph on the same way than the subscribers, and then having it inserted in the
system without any ad-hoc mechanisms.

If the origin of each received data tuples is clearly specified in them, users can know where the relevant
informations comes, and so being informed that a new source has arrived, which mean they could change
their subscriptions if it points out that they could have better results. On the other hand, a clear specification
of the origin of each query could help to maintain an index of the published data. Such an index helps for
prevent the annoyances coming from the churn (a node that was temporary disconnected can ask, when it
come back, to get the data it missed during this time), but could also give some more informations about
the tuples that are available on the sources, allowing to deploy a more classical search engine.

Users that wants to express one-shot queries (having the results once time, without staying in the system)
could so ask the related communities to indicate them where are the relevant sources and what kind of
informations they could give, allowing to easily perform any kind of querying in a fully-open and distributed
system.

9 Conclusion

In the general problem of streams management, especially the publish/subscribe systems, we address in
this paper the construction of an extended query system with the objective to share and limit the network
and computation load in a scalable way, with the condition that users must work on their own interests.

To do this, our proposition is to build a participative system, called $\textit{Q\textendash Tor}$ ($\textit{Query Torrent}$), grouping
subscribers into communities. The principle is to group the users having equivalent subscriptions into
communities, allowing them to share their work, and then to organize those communities, regarding the
rewritibility relations between the queries.

We show in the evaluations that our proposition, applied on several cases of filtering systems, reduces the
global network and processing costs and achieves a good scalability in the organization of large distributed
systems based on the queries.

Setting up a better community organization, based on a multi-layered approach, is a promising future
work. Security aspects will also have to be studied, since users rely on the correct computation of others.
Our current proposition is so only the starting of our work, but we can already point out that our system
gives a lot of interesting possibilities.

References

of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS ’98,

the Thirtieth international conference on Very large data bases - Volume 30, VLDB ’04, pages 456–467.


Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The stanford stream data manager. IEEE Data

Splitstream: High-bandwidth multicast in cooperative. In SOSP ’03: Proceedings of the nineteenth ACM symposium on
Operating systems principles. ACM Request Permissions, December 2003.


[17] Zeinab Hmedeh, Nicolas Travers, Nelly Vouzoukidou, Vassilis Christophides, Cédric Du Mouza, and Michel Scholl. Everything you would like to know about RSS feeds and you are afraid to ask. In BDA’11, Base de Données Avancées, Maroc, October 2011.


[37] Christos Tryfonopoulos, Stratos Idreos, Manolis Koubarakis, and Paraskevi Raftopoulou. Distributed large-scale information filtering.

Contents

1 Introduction 1

2 Related Work 2

3 Theoretical background 5

4 Problem Definition 6
  4.1 Publish/subscribe organization ........................................ 7
  4.2 Systems classification .................................................. 8
    4.2.1 Unicast system .................................................... 8
    4.2.2 Mirrored systems ................................................ 9
    4.2.3 Rewriting systems ............................................... 9
  4.3 Problem statement .................................................... 10
    4.3.1 Evaluation criteria .............................................. 10
    4.3.2 Our objectives .................................................. 11

5 The QTor approach: collaborating communities 11
  5.1 Communities: an abstraction to head QTor ......................... 11
  5.2 Organizing communities using a query hypergraph ................. 12
  5.3 An incremental way to organize the system ....................... 13
  5.4 Subscribers’ collaboration inside a community .................... 14

6 Application 14
  6.1 Strategic choices .................................................... 14
    6.1.1 Community inner organization ................................ 14
    6.1.2 Relation between parent and children communities ............ 15
    6.1.3 Insertion of a new subscriber ................................ 15
    6.1.4 Leaving of a subscriber ....................................... 16
    6.1.5 Moving of an existing community .............................. 16
  6.2 Algorithms .......................................................... 17
    6.2.1 Searching equivalences ........................................ 17
    6.2.2 Selecting the best rewriting .................................. 17
    6.2.3 Inner organization ............................................. 19

7 Evaluation 20
  7.1 Experimental setup .................................................. 20
    7.1.1 Query language ................................................ 20
    7.1.2 Baseline organizations ......................................... 20
    7.1.3 Experimental conditions ....................................... 20
    7.1.4 Future experimentations ....................................... 21
  7.2 Experimental results ................................................ 21
    7.2.1 Organization overhead ......................................... 21
    7.2.2 Query processing ............................................... 21
    7.2.3 Latency ........................................................ 23

8 Future works 23
  8.1 Distributed organization and dynamic aspects .................... 23
  8.2 Reliability, security and access control .......................... 24
  8.3 More uses of the contiguity ........................................ 24
  8.4 Source discovering and one-shot queries .......................... 24

9 Conclusion 25