Using Materialized Views to Enhance a Traceable P2P Record Exchange Framework

Fengrong LI[1] and Yoshiharu ISHIKAWA[2,1,3]
[1]Graduate School of Information Science, Nagoya University
[2]Information Technology Center, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan
[3]National Institute of Informatics, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: lifr@db.itc.nagoya-u.ac.jp, ishikawa@itc.nagoya-u.ac.jp

***Abstract*— P2P technologies are drawing increasing attention nowadays, and have been widely deployed on the Internet for various purposes. Unlike the traditional client-server architecture, a P2P network allows all computers to communicate and share resources as equals and does not depend on a central server for control. In such an environment, tracing how data is copied between peers and how data modifications are performed are not easy because data replications and modifications are performed independently by autonomous peers. This creates inconsistencies in exchanged information and results in a lack of trustworthiness.**

To provide reliable and flexible information exchange facility in P2P networks, we have proposed a framework for enabling *traceable record exchange*. In this framework, a computer can exchange structured records with a predefined schema with other peers. The framework supports a *tracing facility* to query the *lineage* of the records obtained. A tracing query is described in *Datalog* and executed as a recursive query among cooperating peers in a P2P network. In the query execution process, the exchange and modification histories of the queried records are collected dynamically from relevant peers.

In this paper, we focus on how to enhance the traceable P2P record exchange framework using materialized views. First, we discuss how to construct *materialized views* in our framework. Then we present methods for reducing query processing cost and providing *fault tolerance* using the materialized views.

***Index Terms*— information exchange, P2P technologies, data provenance, query processing, materialized views, fault tolerance**

I. INTRODUCTION

Peer-to-peer (*P2P*) technologies already play important roles in supporting flexible information exchange and communications in large-scale networks. Flexible and scalable information exchange has been already realized in P2P file exchange systems such as Gnutella [1], Napster [2], and ICQ [3]. P2P technologies also provide the foundations needed for rich information services as they eliminate the need for a dedicated central server, allowing all computers to communicate and share resources as equals; in other words, all computers in the network act as both servers and clients. Although P2P technologies are already widely used for distributed data storage, file sharing, content delivery, collaborative computing, and so on, there remain some important problems.

One important issue is the *trustworthiness* of the information exchanged in a P2P network. Since duplications and changes to data may be made independently by every peer, it is difficult to follow the path of data through the network. Trustworthiness of P2P information exchange is especially important in scientific research. For example, in bioinformatics, large distributed databases are maintained through the cooperation of independent research organizations, but there are problems due to inconsistencies between different databases. These inconsistencies arise because data may be copied or modified by a researcher or curator and then copied or modified again by another researcher. The notion of *data provenance* (also known as *lineage*) is an important idea for solving this problem [4], [5]. Data provenance tries to provide evidence about how a data item was obtained from other data items and why a specific data item exists in the database.

We have extended the notion of data provenance to information exchange in a P2P network. In this context, we can consider questions related to the trustworthiness of exchanged data, such as “Which peer is the original creator of this data?” and “Who else has a copy of this data?” Such questions can be answered by storing and using the information exchange and modification histories in the peers which participate in the exchange process. We have proposed a *traceable P2P record exchange framework* in which tuple-structured *records* are exchanged [6], [7]. In this framework, records are exchanged among peers and peers can modify, store, and delete their records independently.

An important feature of the framework is that it is based on databases. To ensure traceability, each peer maintains its own relational tables for storing record exchange and modification histories. To make the tracing process easy, the framework provides an abstraction layer which virtually integrates all distributed relations and a *Datalog*-based query language for writing tracing queries in an intuitive manner. Another feature of the framework is that it employs a “*pay-as-you-go*” approach [8] for tracing: the system performs minimum maintenance tasks for tracing and a user pays the cost when he issues a tracing query.

In this paper, we focus on the issue of how to use the notion of a materialized view to improve our P2P

doi:10.4304/jait.2.1.27-39

record exchange framework for determining how data is exchanged among peers and why data is located on a particular peer. One motivation is efficiency: we can use materialized views to speed up query processing. Another essential consideration for a P2P network is reliability—how to cope with system failure and churn. Failure and the unexpected departure of a peer is a critical problem that influences whether a tracing query is executed correctly. We present the key approaches for solving these problems, along with corresponding experimental results.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III describes the outline of the traceable P2P record exchange framework. Section IV gives the definitions of materialized views in the present context. Section V explains how materialized views are used to enhance the underlying framework and discusses the maintenance of materialized views. Section VI presents the experimental results. Finally, Section VII contains our conclusions and considers avenues of future research.

II. Related Work

A. P2P Databases

Although it is still a relatively recent field, P2P database research has experienced rapid growth. There is a variety of research topics, such as coping with heterogeneity, query processing, and indexing methods [9]. Most proposals focus on the heterogeneity of schema and databases. For example, the Piazza system [10] enables the sharing of heterogeneous data in a distributed and scalable manner. In order to process a query issued by a user, the query is reformulated according to mappings so that it can cope with the heterogeneous underlying databases. The query answering system expands the mappings relevant to the query, and retrieves data from other peers. The PeerDB system [11], inspired by information retrieval techniques, creates a kind of data keyword thesaurus to store name mappings and it facilities sharing of data without a global schema. One project that is highly related to our problem is the ORCHESTRA project [12], [13], which aims at the collaborative sharing of evolving data in a P2P network. In contrast to these systems, we focus on supporting reliable and trustful P2P record exchange, based on a simple record exchange scenario, and do not consider schema heterogeneity.

B. Data Provenance

The term *data provenance*, or alternatively *lineage tracing*, refers to the process of tracing and recording the origins of data, transformation of databases, and the movement of data between databases. The target field of data provenance is quite wide and covers data warehousing [14], uncertain data management [15], [16], and scientific fields, such as bioinformatics [17]. In this research area, one well-known project is the *Trio* project, in which both uncertainty and lineage issues are considered [16].

For data provenance, some taxonomies have been proposed. Buneman et al. [18] introduced the notions of *where-provenance* and *why-provenance*. The former is addresses the question “where did the data come from?” and the latter “why does the data exist?” Since our framework treats problems such as ”which peer provided this data?” and ”Did other peers copy this data?”, it belongs to the where-provenance category. Another taxonomy distinguishes between the *lazy* approach and the *eager* approach [5]. The former describes models in which queries for tracing lineage are executed when necessary, and the latter describes the case that metadata or annotations [17] representing lineage are continuously maintained. Our approach to traceability is based on histories maintained at peers and thus belongs to the eager approach.

Data provenance is a common problem that is frequently encountered in databases that undergo many transformations, exchanges, and modifications [5], [19]. However, the notion of data provenance has not previously been applied to P2P information exchange, except in the ORCHESTRA project [12], [13]. However, that project focuses on the schema heterogeneity issue rather then data provenance. In contrast, our research is devoted to data provenance in P2P information exchange, which is quite important for ensuring that the data obtained from the network can be trusted.

C. Dataspace Management

Dataspace management focuses on a highly flexible integration scenario [8]. This is an emerging field of database research and focuses on more flexible information integration over the network. In any applications involving multiple heterogeneous data sources (e.g., personal information management) it may not be necessary or practical to require full integration of information sources beforehand; instead, it may be reasonable to perform information integration dynamically when a user request is issued. Such integration is called the “*pay-as-you-go*” approach [8]. Since our approach focuses on the integration of historical information stored in distributed peers, the “pay-as-you-go” approach is the better choice because it does not interfere with the autonomy of the peers and the tracing requests do not occur often. This approach has an additional benefit that it allows a flexible tracing query representation using the Datalog query language.

D. Declarative Networking

*D*eclarative networking uses a high-level declarative language to express overlay networks in a highly compact and reusable form [20]. Our query processing approach uses the variation of declarative networking described in [21]. As proved in the *declarative networking* project [20], [21], declarative recursive queries are a very powerful tool in writing network-oriented database applications such as sensor data aggregation. In contrast

to their approach, our focus is on compact and understandable tracing query specifications. The objective of our framework is to realize traceable record exchange in a P2P network and is based on the architecture introduced in Section III. Datalog queries are used not only for describing high-level tracing requirements, but also for representing distributed query execution.

E. Materialized Views

Materialized views are snapshots of relational views and can be used to speed-up query processing by precomputing frequently used query results. Efficient maintenance of materialized views is very important in practical databases. A detailed survey of the maintenance of materialized views can be found in [22]. We consider the incremental maintenance of materialized views in the context of deductive databases. Although some incremental materialized view maintenance methods have been already proposed in the literature, there are few papers that consider deductive databases [23]. For maintaining general recursive views incrementally, [24] proposed the *DRed* algorithm that can handle incremental updates. However, as described later, the algorithm assumes a centralized environment, and it is quite costly to apply the algorithm in our context since the maintenance process is propagated among distributed peers.

In our framework, materialized views help to reduce the response time for tracing queries, especially for queries about past histories. So we develop a query processing method which uses materialized views effectively and a view selection and maintenance method which considers the trade-off between cost and benefit. In addition, materialized views can be used as cached data for recovering information when a system failure happens. We also mention how we can use materialized views to provide fault tolerance.

III. System Framework

In this section, we describe why we proposed the traceable P2P record exchange framework and give an overview of its present status.

A. Motivating Example

Consider an application in which autonomous peers share information about novels in a P2P network. Figure 1 shows an instance of a record set `Novel` owned by a peer in the network. The record set consists of four attributes: `title`, `author`, `language`, and `year`. Other peers also maintain their `Novel` records with the same structure, but their contents are not necessarily the same. The record structures shown in this example are so flexible that they can be used for many other tasks such as scientific information exchange. In addition, our framework can be applied to file exchange, where a record can contain metadata for a specific file.

In our record exchange framework, each peer can enrich its own record set by searching for records in the P2P network and incorporate interesting records in its local database. In addition, a peer can modify and delete its own records and can provide them for other peers in the P2P network. A *traceability problem* occurs, for instance, when a peer wishes to ask the question: "Which peer originally created the record `(War and Peace, Leo Tolstoy, Russian, 1865)`?" However, finding such lineage information for data in a P2P network is quite difficult without a supporting facility.

title	author	language	year
Pride and Prejudice	Jane Austen	English	1813
Madame Bovary	Gustave Flaubert	French	1857
War and Peace	Leo Tolstoy	Russian	1865

Figure 1. Example record set `Novel`

B. Traceable P2P Record Exchange Framework

To support the notion of data provenance in P2P information exchange, we proposed the concept of a *traceable P2P record exchange framework* in [6], [7].[1] In the framework, a *record* means a tuple-structured data item that obeys a predefined schema shared globally in a P2P network. We assume that each peer corresponds to a user and maintains the records owned by that user. Every peer can act as a provider and a searcher. Records are exchanged between peers and peers can modify, store, and delete their records independently.

To represent records and their historical information, we employ a layered architecture with different abstraction levels. In the following, we briefly explain the *three-layer model* using an example.

a) User Layer: The *user layer* supports what users see. Each peer has its own record set in the user layer, but their contents are not the same. Peers can behave autonomously and exchange records when required. A peer can find desired records from other peers by issuing a query.

Peer A

title	author
t1	a1
t5	a5

Peer B

title	author
t1	a2
t4	a4

Peer C

title	author
t3	a3
t1	a2

Peer D

title	author
t1	a1
t5	a5

Figure 2. Record sets in user layer

For ease of presentation, we simplify the example shown in Fig. 1 above. Assume that each peer in a P2P network maintains a `Novel` record set that has two attributes, `title` and `author`. Figure 2 shows four

[1]We use "record exchange" to distinguish our problem from *data exchange* [13], which is the problem of taking data that obeys a source schema and creating data under a target schema that reflects the source data as accurately as possible.

record sets maintained by peers A to D in the user layer. Each peer maintains its own records and wishes to incorporate new records from other peers in order to enhance its own record set. For example, the record `(t1, a1)` in peer A may have been copied from another peer and registered in its local record management system.

b) Local Layer: Each peer maintains the minimum amount of information that is required to represent its own record set and local tracing information, which consists of the creation (registration), modification, deletion, and exchange histories related to the peer itself, and which facilitates traceability. All the information required for tracing is maintained by distributed peers in the *local layer*. When a tracing query is issued, the query is processed by coordinating related peers in a distributed and recursive manner. In our framework, every peer maintains the following four relations in its local record management system implemented using an RDBMS.

For example, peer A, shown in Fig. 2, contains the four relations in Fig. 3.

`Data[Novel]@'A'`

id	*title*	*author*
#A1	t1	a2
#A2	t1	a1
#A3	t5	a5

`Change[Novel]@'A'`

from_id	to_id	time
#A1	–	...
#A1	#A2	...
–	#A3	...

`From[Novel]@'A'`

id	from _peer	from _id	time
#A1	B	#B1	...

`To[Novel]@'A'`

id	to_ peer	to_ id	time
#A3	D	#D2	...

Figure 3. Relations in local layer for peer A

The contents and role of each relation shown in Fig. 3 is described as follows:

- `Data[Novel]`: This maintains all the records held by the peer. Every record has a unique record id for maintenance purposes. Note that there are additional records compared to those in Fig. 2; they are *deleted* records and are usually hidden from the user. They are maintained to provide data provenance.
- `Change[Novel]`: This is used to hold the creation, modification, and deletion histories. Attributes `from_id` and `to_id` express the record ids before and after a modification. The attribute `time` represents the timestamp of modification. When the value of the `from_id` attribute is null (−), it means that the record was created at the peer. Similarly, when the value of the `to_id` attribute is null, it means that the record has been deleted.
- `From[Novel]`: This stores the information about records which were copied from other peers. When a record is copied from other peer, the attribute `from_peer` contains the peer name, and the attribute `from_id` has the record's id at the original peer. Attribute `time` stores the timestamp information.
- `To[Novel]`: This relation plays the opposite role to `From[Novel]` and stores information about records were sent from peer A to other peers.

Although `From[Novel]` and `To[Novel]` contain duplicated information, duplicates are stored on different peers. For example, for the tuple of `From[Novel]@'A'` in Fig. 3, there exists a corresponding tuple `(#B1, A, #A1, ...)` in `To[Novel]@'B'`. When the record is registered at peer A, `From[Novel]` at peer A and `To[Novel]` at peer B are updated cooperatively to preserve consistency.

The local layer manages the records and historical information for each peer. This distributed maintenance of lineage preserves independence of peers while supporting various types of tracing queries. When a tracing query is issued, we need to collect the required information from the relevant peers. The record set in the user layer of a peer is just a restricted *view* of its local layer relations that hides lineage information from the user. Figure 4 illustrates the relationship between the user layer and the local layer.

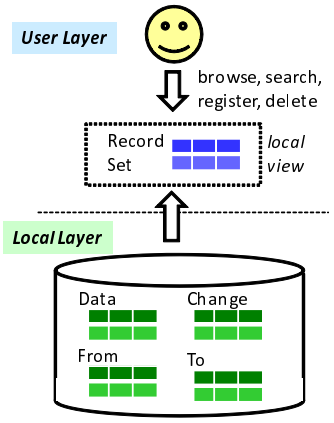

Figure 4. Local layer vs. user layer

c) Global Layer: To aid understanding and simplify the writing tracing queries, we provide an abstraction layer called the *global layer* which integrates all distributed relations virtually and provides a *Datalog*-like query language [25] for writing tracing queries in an intuitive manner. Three virtual global views are constructed by unifying all the relations held by distributed peers. Figure 5 shows three virtual global views for the peers shown in Fig. 2. These global virtual views are used as intuitive images for describing tracing queries.

The `Data[Novel]` view in Fig. 5 unifies all the `Data[Novel]` relations held by peers A to D. The `peer` attribute stores peer names. The combination of a peer name and a record ID ensures that each record is uniquely identified in the entire P2P network. This uniqueness is essential for lineage tracing. `Change[Novel]` is also a global view which unifies all the `Change[Novel]` relations in a similar manner. `Exchange[Novel]` unifies all the underly-

Data[Novel] View

peer	id	*title*	*author*
A	#A1	t1	a2
A	#A2	t1	a1
A	#A3	t5	a5
B	#B1	t1	a2
B	#B2	t4	a4
C	#C1	t1	a1
C	#C2	t3	a3
C	#C3	t1	a2
D	#D1	t1	a1
D	#D2	t5	a5

Change[Novel] View

peer	from_id	to_id	time
A	#A1	–	...
A	#A1	#A2	...
A	–	#A3	...
B	–	#B2	...
C	–	#C2	...
C	#C1	–	...
C	#C1	#C3	...
D	–	#D1	...

Exchange[Novel] View

from_peer	to_peer	from_id	to_id	time
D	C	#D1	#C1	...
C	B	#C3	#B1	...
B	A	#B1	#A1	...
A	D	#A3	#D2	...

Figure 5. Three views in the global layer

ing `From[Novel]` and `To[Novel]` relations in the local layer. Attributes `from_peer` and `to_peer` give the source and the destination of a record exchange, respectively. Attributes `from_id` and `to_id` contain the ids of the exchanged record on both peers.

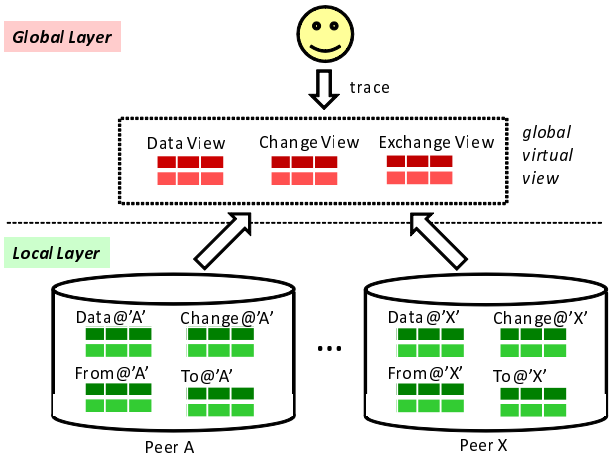

Figure 6. Local layer vs. global layer

The global layer lies over the local layer and provides three global virtual views to allow the user to write tracing queries easily. We can summarize the relationship between the local layer and the global layer in Fig. 6.

C. Query Specification

When a tracing requirement occurs, we need to aggregate the relevant historical information stored in the distributed peers. Since recursive processing is required to collect historical information, our framework provides a modified version of the *Datalog* query language [25]. We now give some tracing query examples.

Example 1: Suppose that peer A holds a record with title `t1` and author `a1` and that peer A wants to know which peer originally created the record. The following query fulfills this requirement:

Query Q1

```
ID(I1) ← Data[Novel]('A', I1, 't1', 'a1'),
ID(I2) ← ID(I1), Change[Novel]('A', I2, I1, _)
BReach(P, I1) ← ID(I2),
                Exchange[Novel](P, 'A', I1, I2, _)
BReach(P, I1) ← BReach(P, I),
                Change[Novel](P, 'A', I1, I2, _)
BReach(P, I1) ← BReach(P, I2),
                Exchange[Novel](P1, P, I1, I2, _)
Origin(P) ← BReach(P, I),
            ¬ Exchange[Novel](_, P, _, I, _)
            ¬ Change[Novel](P, I1, I, _),
            I1 != NULL
Query(P) ← Origin(P)
```

`I` and `P` are variables and '_' indicates an anonymous variable. The relation `ID` defined by the first two rules is used to find the originally assigned ID at the local peer. The relation `BReach` defined by the third and fourth rules means "Backward Reachable". It recursively traverses the arrival path of tuple `(t1, a1)` until it reaches the origin. The fifth rule is used for finally determining the originating peer name—it should be reachable from peer A and should not have received the record from any other peer. The last rule gives the final result expected by the user. □

Example 2: The example shown above focused on backward traversals of lineage information. However, it is also possible to issue queries for forward traversals. Query Q2, which retrieves all the peers which have copied the record (t1, a1) owned by peer A, can be described as follows.

Query Q2

```
Reach(P, I1) ← Data[Novel]('A', I2, 't1', 'a1'),
               Exchange[Novel]('A', P, I2, I1, _)
Reach(P, I1) ← Reach(P, I2),
       Change[Novel](P, I2, I1, _), I1 != NULL
Reach(P, I1) ← Reach(P1, I2),
               Exchange[Novel](P1, P, I2, I1, _)
Query(P) ← Reach(P, _)
```

The first rule is used to find which peers copied record (t1, a1) from peer A directly. The second and third rules retrieve all the peers which copied the record indirectly. After that, relation `Reach` will contain all peer names which copied the target record of peer A. In contrast to query Q1, the query result may change as time passes. □

Note that Queries 1 and 2 perform backward and forward traversals of lineage information, respectively. However, Datalog is so flexible that we can specify various types of queries using the three global views. Please refer to [6], [7] for details.

In our framework, we limit the allowable form of a tracing query. Roughly speaking, a limited class of Datalog¬ programs that are safe, linear, and stratifiable is acceptable. Even with such constraints, we can still preserve the expressibility of tracing queries and can support efficient query execution. The detail is given in [26].

D. Query Processing

In the original framework as described above, each peer only maintains the minimum amount of information that is required to represent its own record set and local tracing information in the local layer. Remember that tracing queries are described in Datalog in terms of virtual views in the global layer. The first step of query processing is to transform the given query to suit the organization of the local layer. This step is performed based on simple transformation rules [26].

Example 3: For example, Query Q1 is mapped as follows:

Mapped Query Q1

```
ID(I1) ← Data[Novel]@'A'(I1, 't1', 'a1'),
ID(I2) ← ID(I1), Change[Novel]'A'(I1, I2, _)
BReach(P, I1) ← ID(I2),
                From[Novel]@A(I2, P, I1, _)
BReach(P, I1) ← BReach(P, I2),
                Change[Novel]@P(I1, I2, _)
BReach(P1, I1) ← BReach(P, I2),
                 From[Novel]@P(I2, P1, I1, _)
Origin(P) ← BReach(P, I),
          ¬ From[Novel]@P(I, _, _, _),
     ¬ Change[Novel]@P(I1, I, _), I1 != NULL
Query(P) ← Origin(P)
```

The symbol '@' is a *location specifier* which indicates the location (peer id) of the relation in the local layer. If a constant peer name follows this symbol such as `@'A'`, it means that the relation is located at peer A. `From[Novel]@P2` represents the `From[Novel]` relation at peer `P2`, where `P2` is a variable representing a peer name. The variable is instantiated while the query is being processed. This query is mainly represented using the `From[Novel]` and `Change[Novel]` relations in the local layer since it detects past histories. □

In the second step of query processing, the mapped query is executed with the cooperation of the relevant peers in the P2P network. The query processing strategy we mainly employed is the *semi-naive method*, which is a common query processing strategy in deductive databases [25]. We extended the semi-naive method to cope with our situation, in which a query is executed in a distributed environment. The detail of the query processing method is given in [26].

Example 4: Figure 7 illustrates how query Q1 is executed for our example. Since we do not have enough space for describing the query processing algorithm, the process is explained intuitively. First, the initial peer A executes the query locally and gets intermediate results `ID`, $\Delta^{\text{new}}_{\texttt{BReach}}$ and $\texttt{BReach}^{\text{new}}$. `ID` is only returned as the ID if peer A created the record in the local database. $\texttt{BReach}^{\text{new}}$ contains the information of the peers which are on the path from peer A to the origin. $\Delta^{\text{new}}_{\texttt{BReach}}$ contains tuples which are new in this iteration step. This drives the query process based on the semi-naive method. Since peer A has reached the fixpoint, it tries to find other peers to continue the query process. In this case, peer B is such a peer—the decision about which peer to choose is made by considering the contents of $\Delta^{\text{new}}_{\texttt{BReach}}$.

After receiving the intermediate results from peer A, peer B starts the local query process and gets new $\Delta^{\text{new}}_{\texttt{BReach}}$ and $\texttt{BReach}^{\text{new}}$. Then the query is forwarded to peer C and the semi-naive query evaluation iterates twice in peer C. Finally, the query is forwarded to peer D. In the query process of peer D, the last two rules are executed because peer D is the origin. In this case, the semi-naive query evaluation iterates twice in peer D and reaches the fixpoint. Since there are no following peers, we terminate the process, and the results are returned back along the forwarding path. □

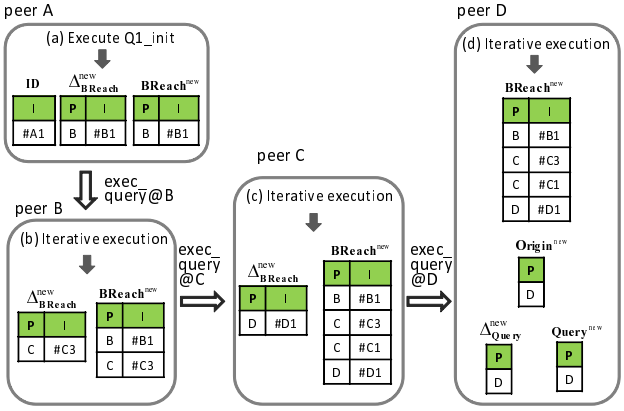

Figure 7. Execution of Query Q1 based on the semi-naive method

This is a rough description of query processing. To understand the remaining parts of this paper, it is not necessary to know the details of query processing. We only need to understand the essential idea: a tracing query is executed by forwarding messages along the record exchange paths and the process is driven by the contents in the database of each peer.

As shown in the example, the query processing is based on the "pay-as-you-go" approach [8]. This means that we need to aggregate the required historical information from the distributed peers when a tracing query is issued by a user; the user should pay the cost when he or she traces information.

E. Problem Statement

One of the problems we consider is *efficiency*. The advantages of the "pay-as-you-go" approach are that it is simple and there is no redundancy with respect to storage cost. However, when we perform query processing, it is necessary to pass the requirement to all the related peers as the process traces the path along which the records were exchanged. Generally, the cost for query processing is relatively large.

Another problem is *fault-tolerance*. Unexpected failures may occur in a P2P network due to network faults and other reasons. In addition, a peer may leave the

network without notice.[2] A failure may affect query processing because we cannot forward the query to a missing peer.

We should construct the framework so that it can automatically recover from partial failures without seriously affecting the overall performance and continue to process a tracing query in an acceptable way. To solve these problems, we consider using materialized views to enhance our traceable P2P record exchange framework.

IV. Constructing Materialized Views

A. Strategies for the Construction of Materialized Views

Materialized views, which can be used to summarize, precompute, and replicate data, play important roles in databases [23]. In our case, we use them to perform data replication to improve the efficiency and the reliability of the framework. We assume that each peer maintains four materialized views: *MVData*, *MVChange*, *MVFrom*, and *MVTo*. These correspond to the Data, Change, From, and To relations in the local layer, respectively.

In the following, we explain the strategies and decisions involved in the construction of materialized views.

1) Each peer maintains four materialized views, but the contents of materialized views located at different peers may be different. For example, materialized view `MVData@'A'` may be different from `MVData@'B'`.
2) Not all of the records are stored in materialized views, but only the exchanged records are stored. For example, suppose that a record, say #A1, was created in a peer A, but it had not been exchanged in the P2P network until now. In this case, other peers do not require the information that record #A1 is recoverable and traceable; only peer A should be responsible for #A1. In contrast, if peer B copied record #A1 from peer A, the copying history is important to peers A and B for tracing, and should be recoverable.
3) The third decision is related to the replication policy. A simple approach is to replicate the data in a peer to some other peer(s). We take a different approach: the peer in which a record is replicated is decided based on the lineage, that is, the way that the record has been copied through the P2P network. This means that different records in the local relations of a peer may be replicated in the materialized views at different peers. As a result, materialized views in one peer may store records incorporated from many peers. As shown below, we can execute tracing queries efficiently based on this policy.

[2]In our scenario, each peer is rather stable in contrast to conventional P2P file exchange. For example, in scientific information exchange, each peer may correspond to a research organization. In this case, an unexpected failure would be a rare event, but it may happen.

B. Target Scope

We assume that tracing queries do not occur frequently so that it is not a wise idea to pay high maintenance costs only for the efficient tracing. In our case, materialized views do not store all of the information in the whole P2P network. They are only used to store information held by the peers in a limited scope.

A *target scope* is determined by the materialized view maintenance policy. The maintenance policy employed in this paper is based on the number of *hops*. In this paper, the number of hops means the number of peers involved in the process of record exchange for the given record. Materialized views in each peer store the related information for up to k hops around it. For example, if peer A received a record from peer B and peer B received the record from peer C, peer C is in two hops from peer A in terms of that record. Thus, if $k = 2$ is used for materialization, materialized views at peer C should store the information that peer A has received the record. In addition, the materialized views at peer A also should store the information that peer C has a copy of the record.

Example 5: Figure 8 shows the target scope of the materialized views for peer X in case of $k = 2$. We assume that the record `(t1, a1)` was originally created by peer D and then peer D published the record to the P2P network. Suppose that some peers copied the data at some moments. A solid arrow in the figure shows the route of the record that has been copied. In this case, peers A, B, I, and E are the peers in the scope of the materialized views at peer X since there were record exchanges between them and peer X is connected directly or indirectly in two hops. Thus, the materialized views MVData, MVChange, MVFrom, and MVTo at peer X should store the related information in the local layer Data, Change, From, and To relations at peers A, B, I, and E. □

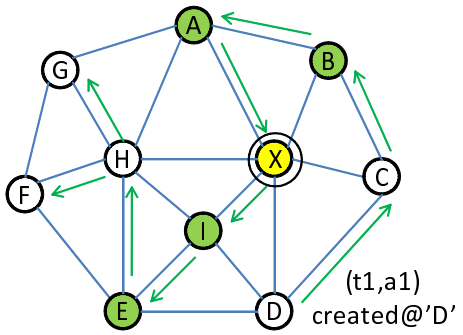

Figure 8. Target scope for peer X ($k = 2$)

C. Definitions of Materialized Views

In the following, we show the representation of four materialized views at peer X for the case of k hops. Like a tracing query, they are expressed in Datalog using the Data, Change and Exchange virtual views in the global layer.

a) MVData: MVData is a materialized view that stores the exchanged records which are in the target scope. MVData stored at peer X can be represented as follows.

Definition of MVData at peer X

```
RData1(P, I1, T, A, H)
                ← Data[Novel]('X', I2, T, A),
          Exchange[Novel](P, 'X', I1, I2, _),
                  H=1
RData1(P, I1, T, A, H)
                ← RData1(P, I2, T, A, H),
                  Change[Novel](P, I1, I2, _)
RData1(P, I1, T, A, H)
                ← RData1(P1, I2, T, A, H1),
         Exchange[Novel](P, P1, I1, I2, _),
                  H=H1+1, H<=k
RData2(P, I1, T, A, H)
                ← Data[Novel]('X', I2, T, A),
          Exchange[Novel]('X', P, I2, I1, _),
                  H=1
RData2(P, I1, T, A, H)
                ← RData2(P, I2, T, A, H),
                  Change[Novel](P, I1, I2, _)
RData2(P, I1, T, A, H)
                ← RData2(P1, I2, T, A, H1),
        Exchange[Novel](P1, P, I2, I1, _),
                  H=H1+1, H<=k
RData(P, I, T, A) ← RData1(P, I, T, A, H)
RData(P, I, T, A) ← RData2(P, I, T, A, H)
Query(P, I, T, A) ← RData(P, I, T, A)
```

In this program, the variable H is used to count the number of hops. The maximum value of H should be set to be equal to k. `RData1` is the collection of the records within k hops related to the copied record owned by peer X. `RData2` stores the information about which peers within k hops copied the records owned by peer X. `RData1` and `RData2` also store the contents of records in these peers. Peer X executes the program and the result of the program (`Query` relation) is finally stored as a materialized view `MVData` at peer X.

Unfortunately, materialized view `MVData@'X'` as constructed above is only effective for backward traversal when we trace the lineage of a record retrospectively. For efficient forward traversal, we also define a program that collects information for the target record for the forward direction. The program is similar to the example above (although it is slightly simpler) so we omit the detail. Finally, the results of the two programs are unioned to get the materialized view `MVData@'X'`.

b) MVChange: This materialized view is used to store the change histories of the exchanged records in the target scope. The definition given is for collecting information by backward traversal. As in the case of MVData, we define a similar program for forward traversal, and the unified query results are stored as `MVChange@'X'`.

Definition of MVChange at peer X

```
RPeer1(P, I1, T, A, H)
                ← Data[Novel]('X', I2, T, A),
          Exchange[Novel](P, 'X', I1, I2, _),
                  H=1
RPeer1(P, I1, T, A, H)
                ← RPeer1(P, I2, T, A, H),
                  Change[Novel](P, I1, I2, _)
RPeer1(P1, I1, T, A, H)
                ← RPeer1(P2, I2, T, A, H1),
         Exchange[Novel](P1, P2, I1, I2, _),
                  H=H1+1, H<=k
RChg1(P, I1, I2, T, H)
                ← RPeer1(P, _, _, _, H),
                  Change[Novel](P, I1, I2, T)
RPeer2(P, I1, T, A, H)
                ← Data[Novel]('X', I2, T, A),
          Exchange[Novel]('X', P, I2, I1, _),
                  H=1
RPeer2(P, I1, T, A, H)
                ← RPeer2(P, I2, T, A, H),
                  Change[Novel](P, I1, I2, _)
RPeer2(P, I1, T, A, H)
                ← RPeer2(P1, I2, T, A, H1),
         Exchange[Novel](P1, P, I2, I1, _),
                  H=H1+1, H<=k
RChg2(P, I1, I2, T, H)
                ← RPeer2(P, _, _, _, H),
                  Change[Novel](P, I1, I2, T)
RChg(P, I, I1, T) ← RChg1(P, I, I1, T, H)
RChg(P, I, I1, T) ← RChg2(P, I, I1, T, H)
Query(P, I, I1, T) ← RChg(P, I, I1, T)
```

Both MVData and MVChange will increase the costs of storage and management for the operation and maintenance of materialized views. However, there are benefits to introducing them: they not only improve query processing efficiency, but they can also be used for the recovery of data lost when a peer suddenly leaves the P2P network.

c) MVFrom: This materialized view stores the information about records which were copied from other peers within k hops. The materialized view `MVFrom` located at peer X can be described as follows.

Definition of MVFrom at peer X

```
ID(I1) ← Data[Novel]('X', I1, T, A),
ID(I2) ← ID(I1), Change[Novel]('X', I2, I1, _)
FromP(P, I1) ← ID(I),
               Exchange[Novel](P, 'X', I1, I, _)
IDP(I1) ← FromP(P, I1),
          Data[Novel](P, I1, T, A)
IDP(I2) ← IDP(I1),
          Change[Novel](P, I2, I1, _)
FromH(P, I, P1, I1, _, H) ← IDP(I),
         Exchange[Novel](P1, P, I1, I, _), H=1
FromH(P, I1, P1, I2, _, H)
                ← FromH(P I, P1, I1, _, H),
                  Change[Novel](P1, I2, I1, _)
FromH(P, I1, P1, I2, _, H)
          ← FromH(P2, I, P, I1, _, H1),
            Exchange[Novel](P1, P, I2, I1, _),
            H=H1+1, H<=k
Query(P, I, P1, I1, _, H)
                  ← FromH(P, I, P1, I1, _, H)
```

MVFrom is effective for tracing records retrospectively in the backward direction. Thus, in contrast to MVData and MVChange, only one program is used to construct the view. As described below, the management cost is negligible (though an additional storage cost is incurred) because path information caching and record insertion to the materialized view are executed only once when the record is exchanged.

d) MVTo: A similar idea can be applied to the To relation. The materialized view `MVTo` stores information about which peers within k hops copied records from peer X.

Definition of MVTo at peer X

```
ToP(P, I1) ← Data[Novel]('X', I2, T, A),
             Exchange[Novel]('X', P, I2, I1, _),
ToP(P, I2) ← ToP(P, I1),
             Change[Novel](P, I1, I2, _), I2 != NULL
ToH(P, I, P1, I1, _, H) ← ToP(P, I),
             Exchange[Novel](P, P1, I, I1, _),H=1
ToH(P1, I1, P2, I2, _, H)
           ← ToH(P, I, P1, I1, _, H1)
             Exchange[Novel](P1, P2, I1, I2, _),
             H=H1+1, H<=k
Query(P, I, P1, I1, _, H)
                     ← ToH(P, I, P1, I1, _, H)
```

In contrast to MVFrom, the management cost of MVTo is not negligible. This is because we need to deal with future events for the management of MVTo whereas only past events are stored in MVFrom. For example, in Fig. 8, when the record is copied from peer I to peer E, it is necessary to notify peer X about the copy event. In other words, not only peer I and E but also peer X is involved in the transaction of copying the record from peer I to peer E. This introduces an additional overhead to a certain extent.

Finally, we mention our decision rule for the parameter k, which determines the policy of materialized view maintenance. In our approach, k is initially fixed to some value (e.g., $k = 2$), when P2P record exchange is started. An alternative strategy would be to treat k as a variable, allowing different k values to be selected for different peers. This option is interesting, especially when some peers have large storage and high processing power. However, to simplify the algorithms, we do not consider this option and leave the problem for the future.

V. Using Materialized Views for Efficiency and Fault-Tolerance

Materialized views play an essential role for efficient query processing in current database systems. In this section, we first show how we can use materialized views for efficient query processing in our framework. Next, we discuss how we can use materialized views to cope with failures. Materialized views can be considered as replicated data so that we can utilize them when some peer fails. Finally, we discuss how to maintain materialized views in a consistent manner.

A. Query Processing with Materialized Views

For a peer, say X, four materialized views `MVData@'X'`, `MVChange@'X'`, `MVFrom@'X'`, and `MVTo@'X'` are locally stored as base relations. To improve query performance using them, we perform query rewriting. We illustrate the outline of this procedure using an example.

Example 6: Based on the materialized views, we can rewrite the Example Query Q1 in Section III as follows.

Mapped Query Q1

```
ID(I1) ← Data[Novel]@'A'(I1, 't1', 'a1'),
ID(I2) ← ID(I1), Change[Novel]'A'(I1, I2, _)
BReach(P, I1) ← ID(I2),
                From+[Novel]@A(I2, P, I1, _)
BReach(P, I1) ← BReach(P, I2),
                Change+[Novel]@P(I1, I2, _)
BReach(P1, I1) ← BReach(P, I2),
                 From+[Novel]@P(I2, P1, I1, _)
Origin(P) ← BReach(P, I),
          ¬ From+[Novel]@P(I, _, _, _),
     ¬ Change+[Novel]@P(I1, I, _), I1 != NULL
Query(P) ← Origin(P)
```

This query finds the origin of a target record. The differences from the original mapping shown in Example 3 are in the third to sixth rules. In these rules, the predicate names `From` and `Change` are replaced by `From+` and `Change+`, respectively. From+ represents a relational view obtained by the union of the From relation and the MVFrom materialized view. For example, `From+` for peer X is defined by

```
From+[Novel]@X := From[Novel]@X(I1, P, I2, T)
                ∪ MVFrom[Novel]@X(I1, P, I2, T).
```

`Change+` and other views are defined in a similar manner. □

The motivation for this query rewriting is to speed up the process of backward traversal. The query rewriting is easy: as shown in the example, we simply replace predicates `Data`, `Change`, `From`, and `To` to `Data+`, `Change+`, `From+`, and `To+`.[3] For this example, we can skip some peers and can reduce the number of messages exchanged between peers while query processing by using MVFrom and MVChange. The query is executed based on the method described in Subsection III-D; we do not need to modify the query processing strategy.

Example 6: (continued) Figure 9 illustrates the query execution process for processing query Q1 using the materialized views. If we use the original query without materialized views, the query is forwarded between peers in a step-by-step manner as $A \rightarrow B \rightarrow C \rightarrow D \cdots$. In contrast, the modified query using materialized views is forwarded as $A \rightarrow D \rightarrow G \rightarrow \cdots$.

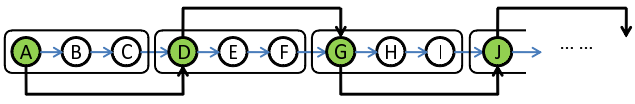

Figure 9. Processing Query Q1 using materialized views ($k = 2$)

In this example, we can skip peers B and C because the information for backward traversal using the information of B and C is stored in `MVChange@'A'` and `MVFrom@'A'`. Thus, instead of query forwarding, peer A uses the information in the materialized views and executes the query locally until it reaches the fixpoint, then decides the next peer (peer D in this case) to which the query should be forwarded. Then, peer D follows a

[3]Strictly speaking, there is no gain from using `Data+[Novel]@'A'` and `Change+[Novel]@'A'` if we consider the semantics of the query. However, since a union operation can be efficiently processed when we issue an SQL query to the underlying DBMS, the overhead is negligible.

similar strategy and skips peers E and F. In this way the query processing is accelerated. □

Next we give another example.

Example 7: Query 2 can be transformed using the materialized views as follows. In this example, peer A wants to detect which peer copied the record (t1, a1) provided by peer A. The rewriting strategy is same as that for Query Q1.

Mapped Query Q2

```
Reach(P, I1) ← Data+[Novel]@'A'(I2, 't1', 'a1'),
               To+[Novel]@'A'(I2, P, I1, _)
Reach(P, I1) ← Reach(P, I2),
       Change+[Novel]@P(I2, I1, _), I1 != NULL
Reach(P, I1) ← Reach(P1, I2),
               To+[Novel]@P1(I2, P, I1, _)
Query(P) ← Reach(P, _)
```

The query process is illustrated in Fig. 10. In our original approach without materialized views, the query processing starts at peer A and the query fragments generated at peer A are first forwarded to peers B and C, and then peer B forwards them to peer D and peer C forwards them to peers E and F, and so on. The query is executed in this way until it reaches the ends of forward paths.

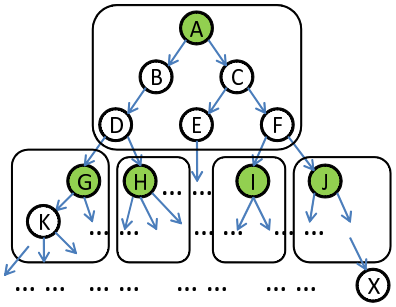

Figure 10. Processing Query Q2 using materialized views ($k = 2$)

When the materialized views are used, peer A can perform more of the query processing locally without communicating B, C, D, E, and F since they are in the target scope of peer A in terms of this record. Then the query execution is continued by peers G, H, I and J. In this way, the number of peers involved in the query process is greatly reduced. □

These two examples indicate that materialized views can be used to speed up query processing. Although they are special examples, they do illustrate our main idea of utilizing materialized views for efficient traversal over record exchange paths. In our framework, forward and backward traversals over record exchange paths occur often, and their efficient processing is quite important. Thus, by using materialized views, we can greatly improve performance.

B. Achieving Fault Tolerance Using Materialized Views

One important issue for executing a tracing query in a decentralized and autonomous P2P network is fault tolerance. If a peer which is located in a record exchange path for the given query leaves the network, the query execution cannot continue and will lead to an incorrect result. However, we constructed materialized views with redundancy. So, even if a peer leaves a P2P network due to some failure, we can recover the part of its contents which is used for lineage tracing by collecting information from related peers.

For recovery purposes, we assume that a peer (say X) specifies one peer (say Y) as its *backup peer*. The backup peer Y maintains the information required for starting the recovery process for peer X. The backup peer Y contains a special relation `Friend@'Y'(P1, P2)`, which contains tuples with the form `('X', P)`. This represents the information that peer X has exchanged some record with peer P. We call P a *friend* of X. When peer X specifies peer Y as its backup peer and Y accepts the offer, the friend history of X is continually managed in peer Y.

We illustrate the process of data recovery using an example.

Example 8: Suppose that peer X in Fig. 8 suddenly leaves the P2P network due to a failure. Let the backup peer of X be peer Y. We assume that the information that peer Y is the backup peer of peer X is stored in a distributed hash table index maintained in the P2P network, and assume that some peer (say Z) is elected to cover the role of peer X for tracing.

To recover lineage information which was stored in peer X, peer Z executes the following program.[4] This program is written at the local layer level.

Recovery Program for Peer X

```
From[Novel]@'X'(I1, P, I2, T) ←
               Friend[Novel]@'Y'('X', F),
               To[Novel]@F(I2, 'X', I1, T)
To[Novel]@'X'(I1, P, I2, T) ←
               Friend[Novel]@'Y'('X', F),
               To[Novel]@F(I2, 'X', I1, T)
Data[Novel]@'X'(I, T, A) ←
               From[Novel]@'X'(I, P, _, _),
               MVData[Novel]@P('X', I, T, A)
Data[Novel]@'X'(I, T, A) ←
               To[Novel]@'X'(I, P, _, _),
               MVData[Novel]@P('X', I, T, A)
Change[Novel]@'X'(I1, I2, T) ←
            From[Novel]@'X'(I, P, _, _),
            MVChange[Novel]@P('X', I1, I2, T)
Change[Novel]@'X'(I1, I2, T) ←
            To[Novel]@'X'(I, P, _, _),
            MVChange[Novel]@P('X', I1, I2, T)
```

The first rule recovers the information of the `From[Novel]` relation originally stored in peer X. It uses the information of the `Friend[Novel]` relation in the backup peer Y to find friends of peer X. If a friend of X sent a record in the past, its `To[Novel]` relation contains the history of the event. In this rule, we collect the information needed to recover `From[Novel]@'X'` from the friends. Similarly, the second rule recovers `To[Novel]@'X'`. The third and fourth rules are for recovering the `Data[Novel]@'X'`

[4]Strictly speaking, this is not a valid program. As shown in Queries Q1 and Q2, valid programs should contain a rule which derives a `''Query''` relation, which is the result of the Datalog program execution.

relation. The third rule uses the `From[Novel]@'X'` relation which was just recovered. This rule utilizes the fact if peer X copied a record with id I from peer P, then `MVData[Novel]` at peer P holds the backup data of `Data[Novel]@'X'`. The fourth rule follows the same pattern but uses the `To[Novel]@'X'` relation. Similarly, `Change[Novel]@'X'` is recovered using the fifth and sixth rules. The four recovered relations are finally stored in peer Y.

In our framework, we assume that a global name service which maps a peer name into its IP address is available. After the recovery is done, we modify the mapping so that a message to peer X is forwarded to peer Y. Thus, peer Y can behave as if it is peer X, and then the tracing facility will work correctly. □

We also recover the four materialized views (e.g., `MVData[Novel]@'X'`) which were originally stored in peer X, although this is not discussed here. This is not difficult because we have already recovered the four base relations of peer X.

In this subsection, we have described a recovery method to cope with a failure. As mentioned previously, we consider that the sudden departure of a peer is a rare event in our framework. We assume that each peer usually follows a protocol when it leaves the P2P network. If peer X leaves, it selects a backup peer Y. After Y copies the information related to tracing from X, peer X can leave the network without introducing a tracing problem.

C. Maintenance of Materialized Views

View maintenance means the process for updating materialized views in response to changes in the underlying database. As we described above, materialized views can speed up query processing greatly and can provide fault tolerance, but they have to be kept up to date. If some of the base relations are changed, materialized views must be updated to ensure correctness. For maintaining general recursive views in deductive databases, several methods have been proposed. For example, [24] presents the *DRed* (Delete and Rederive) algorithm that can handle incremental updates. However, the algorithm assumes a centralized environment, and it is quite costly to apply the algorithm in our context because the maintenance process is propagated among distributed peers.

In our case, we can utilize the feature of our framework that every update can be handled as a tuple insertion; when we delete a record, we do not delete its corresponding tuple in the database but insert a tuple to indicate that the tuple was deleted. If database updates do not involve tuple deletion and modification, the view update problem becomes easier.

Example 9: Consider the case that materialized views are created with the parameter $k = 2$. Assume that record #X1 in peer X is a copy of record #Y1 in peer Y, and record #Y1 is a copy of record #Z1 in peer Z. We consider the updating of record #X1 in peer X. Since $k = 2$, peer Y contains the information for record #X1 in its materialized views `MVData@'Y'`, `MVChange@'Y'`, `MVFrom@'Y'`, and `MVTo@'Y'`. In addition, peer Z contains similar information in its materialized views. Depending on the update type, a new tuple is inserted in each of the following local relations and materialized views:

- update of record #X1: `Data@'X'`, `Change@'X'`, `MVData@'Y'`, `MVData@'Z'`, `MVChange@'Y'`, and `MVChange@'Z'`
- deletion of record #X1: `Change@'X'`, `MVChange@'Y'`, and `MVChange@'Z'`
- copy of record #X1 to other peer W: `Data@'W'`, `To@'X'`, `From@'W'`, `MVData@'X'`, `MVData@'Y'`, `MVTo@'Y'`, `MVTo@'Z'`, and `MVFrom@'W'`.

This means that there are only *insertions* in the database updates in our framework. □

Materialized view maintenance for the insertion case is fairly easy. We run the semi-naive method until the fixpoint. When we reach the fixpoint, the current incremental maintenance is finished [24].

Finally, we should mention the additional care that must be taken with Friend relations when a copy event happens. Consider the copy event shown in the example above: A tuple `Friend('X', 'W')` is inserted in the Friend relation in the backup peer of X, if the tuple is not already in that relation. Similarly a tuple `Friend('W', 'X')` is inserted in the Friend relation in the backup peer of W.

VI. Experiments and Discussions

The purpose of the experiments is to observe the benefit of using materialized views in a simple P2P record exchange model. We used the two queries in Section III and evaluate the query processing cost for different values of parameter k.

The simulation model is summarized as follows. We initially create $N = 100$ peers and $M = 1$ record in each peer. After that, we perform an iteration: we randomly select a peer X, and peer X randomly selects another peer Y; then peer X selects one record randomly from Y and copies it into its local database. We terminate the iteration when the average length of copy paths reaches a specified threshold value. After this preparation is finished, we execute tracing queries.

In this simple scenario, we do not consider record creation, modification, or deletion. The reason is that, although such behaviors are important to modeling the actual behavior of our framework, they do not directly influence the performance of query processing for tracing queries. The major factor is the length of a record copy path, which directly influences the forward and backward traversal cost.

Figure 11 shows the results when Query Q1 is executed based on the strategies shown in the former section. The performance is measured by the number of peers which are involved in the query processing. We compared the performance of three approaches. The basic framework is the baseline method without materialized views. The enhanced frameworks ($k = 1, 2$) are based on the proposed method with the corresponding parameter setting.

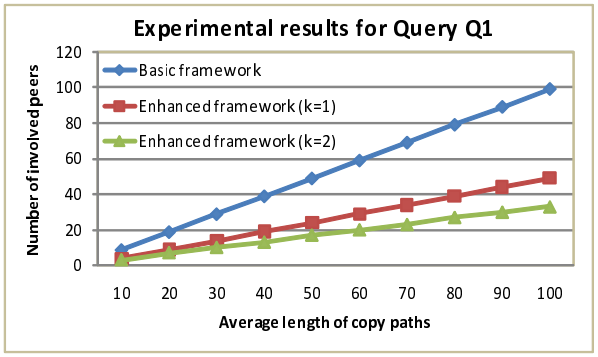

Figure 11. Query processing for Query Q1 (backward traversal)

The experimental results indicate that the use of materialized views actually contributes to skipping peers. The performance for $k = 2$ is better than that for $k = 1$ as we expected. Although we did not measure the actual processing time since the factor is highly dependent on the system setting, we think that the number of peers involved in query processing directly influences the query processing time. The reason is that the amount of communication between peers and the total number of queries issued in the peers are tightly connected with the number of peers involved, and they are the major factors of the query processing cost.

Figure 12 shows the experimental results for Query Q2. The overall behavior is similar to that of Query Q1. The difference is that the overall query processing cost is generally larger than that for Query Q1. This is because query processing for forward traversal needs to follow many branches of copy paths. From this experiment, we can see that using the materialized views greatly reduced the query processing cost.

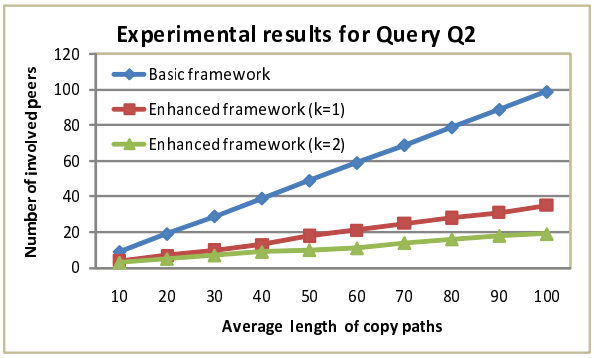

Figure 12. Query processing for Query Q2 (forward traversal)

We omitted the experimental result for storage cost, but the behavior of the proposed method is obvious. If we use the parameter setting $k = 1$, the storage cost roughly becomes $3S$ compared to the normal storage cost S. This is because we need to maintain additional two copies of lineage information for each record in the forward and backward directions. If $k = 2$ is used, the storage cost will reach $5S$ because we need to consider records within two hops. Note that these estimates are maximal because we do not need to manage the information for a record which is not exchanged in the P2P network.

As described above, the parameter k has a strong influence on the storage cost. Although the improvement of query processing time appeals, you may say that the selection of $k = 2$ is not a good idea, considering the storage cost. As an alternative strategy, we can use different parameter settings for MVData, MVChange, MVFrom, and MVTo. For example, we can use $k = 1$ for MVData and MVChange and use $k = 2$ for MVFrom and MVTo. In this case, we can save the storage costs for MVData and MVChange but still support fault tolerance. This setting also assures the validity of query results and is effective for queries which only access From or To relations and do not access MVData and MVChange. We leave this alternative strategy evaluation to future work.

We must also mention the maintenance cost. The evaluation of the maintenance cost is not easy because it also depends on the system settings and the environment. Roughly speaking, the maintenance cost mainly consists of the cost of distributed transactions. Recall the example shown in Subsection V-C. If we update record #X1 in peer X, we need to perform a distributed transaction among peers X, Y, and Z. If we use a large k value, the communication cost may not be negligible. In the situation where record exchanges occur quite frequently, we may need to select a small k value for efficiency. Although our framework incurs a maintenance cost, it would be more efficient than the centralized approach in which all the histories are maintained in one or more servers. In such a case, the processing cost for query processing and maintenance would become the bottleneck for the whole system.

Finally, we consider the fault tolerance issue. Our method can cope with the failure of one peer. If multiple failures occur simultaneously, we may not be able to recover the lineage information for tracing. For example, when two related peers suddenly leave the network at the same time, especially when a peer and its backup peer leave together, it is hard to recover the correct lineage. However, we assume that such events are quite rare in our context. We would like to leave the treatment of such a rare case for future research.

VII. Conclusions and Future Research

For efficient query processing, data replication and caching are popular techniques. Taking into account the practical requirements of tracing, we added additional features to our traceable P2P record exchange framework. Although the storage and maintenance cost will increase, the query processing cost can be reduced and failure of peer can be overcome.

We have described how to define materialized views, how to use them to improve query processing performance, and how to cope with failures. A method for maintaining materialized views was also given. Nevertheless much work remains to be done. In particular, we need to consider the trade-off between query processing cost and

maintenance cost when evaluating the total cost reduction. There are several further issues for future research which can be summarized as follows.

- Enhancement of query expression power: We need to enhance the strategies to handle more complex tracing queries (e.g., tracing queries that involve aggregation requirements). The effectiveness and limitations of the declarative language-based approach will become clearer.
- Efficient coupling with DBMSs: For implementing our framework, we assume that a local record management system in each peer is implemented using a conventional RDBMS. We would like to use more effectively powerful and robust DBMS functionalities that can come from the tight coupling of the record management system and the underlying RDBMS.
- Prototype system implementation and experiments: We are currently developing a prototype system of our P2P record exchange framework, and we have also designed a P2P network simulator that can be used for simulating our prototype system as a virtual P2P network. These developments will have a positive feedback and help to improve our fundamental framework.

ACKNOWLEDGMENTS

This research was partly supported by Grants-in-Aid for Scientific Research (#21013023, #22300034) from the Japan Society for the Promotion of Science (JSPS).

REFERENCES

[1] "Gnutella," http://en.wikipedia.org/wiki/Gnutella.
[2] "Napster," http://en.wikipedia.org/wiki/Napster.
[3] "ICQ," http://www.icq.com.
[4] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren, "Curated databases," in *Proc. ACM PODS*, 2008, pp. 1–12.
[5] W.-C. Tan, "Research problems in data provenance," *IEEE Data Engineering Bulletin*, vol. 27, no. 4, pp. 45–52, 2004.
[6] F. Li, T. Iida, and Y. Ishikawa, "Traceable P2P record exchange: A database-oriented approach," *Frontiers of Computer Science in China*, vol. 2, no. 3, pp. 257–267, 2008.
[7] F. Li and Y. Ishikawa, "Traceable P2P record exchange based on database technologies," in *Proc. APWeb*, ser. LNCS, vol. 4976, 2008, pp. 475–486.
[8] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," in *Proc. ACM PODS*, 2006, pp. 1–9.
[9] K. Aberer and P. Cudre-Mauroux, "Semantic overlay networks," in *VLDB*, 2005, (tutorial notes).
[10] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The piazza peer data management system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 787–798, 2004.
[11] J. Goldstein and P.-A. Larson, "PeerDB: A P2P-based system for distributed data sharing," in *Proc. ICDE*, 2003, pp. 633–644.
[12] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen, "ORCHESTRA: Facilitating collaborative data sharing," in *Proc. ACM SIGMOD*, 2007, pp. 1131–1133.
[13] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir, "ORCHESTRA: Rapid, collaborative sharing of dynamic data," in *Proc. Conf. on Innovative Data Systems Research (CIDR 2005)*, 2005, pp. 107–118.
[14] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," in *Proc. VLDB*, 2001, pp. 471–480.
[15] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom, "ULDBs: Databases with uncertainty and lineage," in *Proc. VLDB*, 2006, pp. 953–964.
[16] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," in *Proc. Conf. on Innovative Data Systems Research (CIDR 2005)*, 2005, pp. 262–276.
[17] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "An annotation management system for relational databases," in *Proc. VLDB*, 2004, pp. 900–911.
[18] P. Buneman, S. Khanna, and W.-C. Tan, "Why and where: A characterization of data provenance," in *Proc. ICDT*, ser. LNCS, vol. 1973, 2001, pp. 316–330.
[19] ——, "Data provenance: Some basic issues," in *Proc. of 20th Conf. on Foundations of Software Technology and Theoretical Computer Science (FST TCS 2000)*, ser. LNCS, vol. 1974, New Delhi, India, Dec. 2000, pp. 87–93.
[20] "P2: Declarative networking," http://p2.berkeley.intel-research.net/.
[21] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: Language, execution and optimization," in *Proc. ACM SIGMOD*, 2006, pp. 97–108.
[22] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, pp. 3–18, 1995.
[23] A. Gupta and I. S. Mumick, Eds., *Materialized Views*. MIT Press, 1999.
[24] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proc. ACM SIGMOD*, 1993, pp. 157–166.
[25] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
[26] F. Li and Y. Ishikawa, "Query processing in a traceable P2P record exchange framework," *IEICE Transactions on Information and Systems*, vol. E93-D, no. 6, pp. 1433–1446, 2010.

Fengrong Li is a Ph.D. candidate majoring in Systems and Social Informatics at the Graduate School of Information Science at Nagoya University, Japan.

Her main research interests lie in data provenance, P2P databases, integration of distributed heterogeneous information, and information retrieval. She is a student member of DBSJ, IPSJ, and IEICE.

Yoshiharu Ishikawa is a Professor at the Information Technology Center, Nagoya University, Japan. He is also a Visiting Professor at the National Institute of Informatics, Japan.

His research interests include spatio-temporal databases, mobile databases, P2P databases, data mining, information retrieval, and Web information systems. He is a member of the Database Society of Japan, IEICE, IPSJ, JSAI, ACM, and IEEE.