Traceable Peer-to-Peer Record Exchange

Fengrong Li
supervised by Yoshiharu Ishikawa
Graduate School of Information Science
Nagoya University, 464-8601, Japan
lifr@db.itc.nagoya-u.ac.jp

ABSTRACT

Peer-to-peer (*P2P*) technology allows us flexible information sharing and communications in a wide-spread network. Unlike the traditional client-server architecture, a P2P network enables a peer to publish information and share data with other peers without central server control. In such an environment, tracing how data is copied between peers and how data modifications are performed is not easy because data replications and modifications are performed independently by autonomous peers. This brings inconsistency in exchanged information and results in lack of reliability.

To provide reliable and flexible information exchange facility in P2P networks, I am working on a framework for *traceable record exchange* in a P2P network. In this framework, a peer can exchange structured records with a predefined schema among other peers. The framework supports a *tracing facility* to query the *lineage* of the obtained records. 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 the related peers. In this paper, the background, the motivation, the outline of the approach, and the current on-going work are presented.

1. INTRODUCTION

In recent years, *peer-to-peer* (*P2P*) *technologies* have attracted much attention [3]. Although existing file exchange services proved the potential effectiveness of P2P technologies by showing their flexibility and scalability, several important issues occurred such as copyright violation and exchange of unreliable information. Development of reliable information exchange in a P2P network has been one of the important issues. One approach to cope with the problem is to support *lineage tracing* facilities [15, 16], which tries to give us evidences how a data item was obtained from other peers and why a data item exists in a peer.

Based on the background, we proposed a network-wide system framework for *traceable P2P record exchange* [10, 12]. We have extended the notion of lineage tracing to information exchange in a P2P network. A *record*, in our framework, is a tuple-structured data item that obeys a predefined schema globally shared in the network. Records are exchanged between peers and peers can modify, store, and delete their records independently. Record exchange and modification histories are maintained in a database of each peer in a distributed manner. The main objective of our work is to realize the *traceability* facility based on the historical information maintained in peers. Using histories, a user can know, for example, which peer initially created a specific record, and how the record was exchanged between peers until it reached the current peer. Since a user can understand why and how a record exists in his or her peer, the user can rely on the information contained in the record. An important feature of our P2P record exchange architecture is that it is based on the *database technologies* to support record exchange and traceability.

A user can trace the lineage of a target record by issuing a tracing query. For helping a user to write a recursive tracing query, the framework provides an abstraction layer which virtually integrates all distributed relations and a *datalog*-based query language for describing tracing queries in an intuitive manner. Another feature of the framework is that it employs "*pay-as-you-go*" approach [9] for tracing. Given a tracing query, distributed peers cooperatively answer it by integrating historical data maintained by them. Such a query can be expressed by a recursive query that traverses peers in the P2P network. The framework provides two query execution modes, *ad-hoc queries* and *continual queries*, for fulfilling different tracing requirements.

This paper gives an overview of our contributions. Section 2 reviews the related work. Section 3 describes the traceable P2P record exchange framework. Section 4 introduces definition of the tracing queries and two execution modes. Section 5 provides the query processing strategies for ad-hoc queries and continual queries. Section 6 describes an on-going work to enhance the framework by using materialized views. Finally, Section 7 concludes the paper.

2. RELATED WORK

There are a variety of research topics on P2P databases, such as coping with heterogeneity, query processing, and indexing methods [1]. The most related field to our research is *information integration* in a P2P network. Record exchange in our framework can be considered as a special type of information integration, in which information (records)

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

are loosely but cooperatively shared in the network. In this sense, a highly related project is ORCHESTRA [7], which aims at collaborative sharing of evolving data in a P2P network. However, our framework is totally different from P2P information integration because we do not try to collect all the information in the P2P network. The key point is that we can trace all the histories about exchanged records if we want using tracing queries.

A related topic is *data provenance*. The target field of that is quite wide and it covers data warehousing [6], uncertain data management [16], database curation [5], and other scientific fields such as bioinformatics [4]. Historical information to support data provenance is often called *lineage*. In our framework, exchange and modification histories stored in peers correspond to lineage to explain how records are obtained and modified in the P2P network. In typical implementation, lineage information is attached to a data item and modified when the item is updated. In contrast, lineage information in our framework is scattered in the P2P network and collected when it is required.

Another related concept is *dataspace system* [9]. It focuses on a highly flexible integration scenario. In some application situations, it is not reasonable to integrate all the available information beforehand. For example, a personal information management system does not necessarily require full integration of information sources. It may be reasonable to perform integration dynamically when a user request is issued. Such integration is called the "*pay-as-you-go*" approach [9]. Since we can assume tracing requirements do not occur quite often, the "pay-as-you-go" approach will be a better choice. It does not highly interfere with the autonomity of peers.

A tracing query in our framework is represented as a recursive query executed in a P2P network. To describe a tracing query in a user-friendly manner, we use *datalog*, which is a well-known query language for deductive databases [2]. The most related work to our research is *declarative networking* [14]. In their research, a datalog-based recursive query processing framework is used for collecting information from P2P networks and sensor networks. In contrast to declarative networking, our framework has the following features: 1) The objective of our framework is to realize traceable record exchange in a P2P network and is based on the architecture introduced in Section 3. Datalog queries are used not only for describing high-level tracing requirements, but also for representing distributed query execution. On the other hand, declarative networking focuses on continual monitoring in a distributed network and does not have the high-level abstraction feature. 2) Declarative networking mainly focuses on continual queries because their target is continual monitoring in a network, but our framework considers ad-hoc queries in addition to continual queries. 3) We use datalog$^\neg$ for representing tracing requirements. By allowing the use of negations, we can greatly improve the query expression power. We carefully restrict allowable tracing queries and effectively use the insertion-only feature of the underlying databases, and construct a clear query processing framework.

3. TRACEABLE P2P RECORD EXCHANGE

In this section, we provide an overview of *traceable P2P record exchange framework* [10, 12] in which tuple-structured *records* are exchanged in a P2P network.[1] The framework consists of the following *three layers*—the user layer, the local layer, and the global layer. In the following, we describe the role of each layer with examples. We assume that information about novels is exchanged among peers in a P2P network.

3.1 User Layer

A peer in our framework corresponds to a user and maintains the records owned by the user. Roughly speaking, the *user layer* supports what a user sees. A peer can create, delete, modify, and register records in its record set based on the peer's decision. Peers can behave autonomously and exchange records when it is required. In addition, a peer can find desired records from other peers by issuing a query.

We assume that each peer in a P2P network maintains a `Novel` record set that has two attributes `title` and `author`. Figure 1 shows four 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 peer B and registered in peer A's local record management system.

Peer A

title	author
t1	a1
t2	a3

Peer B

title	author
t1	a1
t4	a4

Peer C

title	author
t1	a1

Peer D

title	author
t1	a1

Figure 1: Record sets in user layer

3.2 Local Layer

Each peer maintains its own relational tables for storing record exchange and modification histories and facilitates *traceability*. For example, peer A shown in Fig. 1 contains the four relations shown in Fig. 2 in the local layer.

`Data[Novel]@'A'`

id	*title*	*author*
#A1	t1	a1
#A2	t2	a2
#A3	t2	a3

`Change[Novel]@'A'`

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

`From[Novel]@'A'`

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

`To[Novel]@'A'`

id	to_ peer	to_ id	time
#A1	C	#C1	...

Figure 2: Relations in local layer for peer A

The roles of four relations are described as follows:

- `Data[Novel]`: It maintains all the records held by the peer. Every record has its own record id for the maintenance purpose. Each record id should be unique in

[1]We use the term "record exchange" differently from *data exchange* in [7]. The latter 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.

the entire P2P network. Even the *deleted* and *modified* records, which are hidden from the user, are also maintained for lineage tracing.

- `Change[Novel]`: It is used to hold the creation, modification, and deletion histories. Attributes `from_id` and `to_id` express the record ids before/after a modification. Attribute `time` stores the timestamp information. When the value of the `from_id` attribute is the null value (−), it represents that the record has been created at the peer. Similarly, when the value of the `to_id` attribute is the null value, it means that the record has been deleted.

- `From[Novel]`: It records which records were copied from other peers. When a record is copied from other peer, attribute `from_peer` contains the peer name and attribute `from_id` has its record id at the original peer.

- `To[Novel]`: It plays an opposite role of `From[Novel]` and stores information which records were sent from the current peer to other peers.

Note that `From[Novel]` and `To[Novel]` contain duplicates, but they are stored in different peers. For example, for the first tuple of `From[Novel]@'A'` in Fig. 2, there exists a corresponding tuple `(#B1, A, #A1, ...)` in `To[Novel]@'B'`. When the record is registered at peer A, `From[Novel]@'A'` and `To[Novel]@'B'` are updated cooperatively to preserve the consistency.

In the local layer, all the information required for tracing is maintained in peers in a distributed manner. When a tracing query is issued, we need to collect the required information from the related peers. The record set in the user layer of a peer is just a restricted *view* of its local layer relations. Figure 3 illustrates the relationship between the user layer and the local layer.

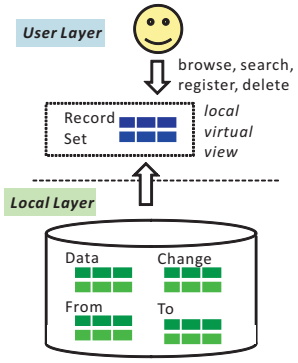

Figure 3: Local layer vs. user layer

3.3 Global Layer

For ease of understanding and writing tracing queries, we provide an abstraction layer called the *global layer* which virtually integrates all distributed relations in the local layer. Figure. 4 shows three virtual global views for peers A to D shown in Fig. 1. Note that we do not materialize the three views shown in Fig. 4. The global virtual views are just used as intuitive images for describing tracing queries.

`Data[Novel]` view in Fig. 4 expresses a view that unifies all the `Data[Novel]` relations in peers A to D shown in

`Data[Novel]` View

peer	id	*title*	*author*
A	#A1	t1	a1
A	#A2	t2	a2
A	#A3	t2	a3
B	#B1	t1	a1
B	#B2	t4	a4
C	#C1	t1	a1
C	#C2	t5	a5
D	#D1	t1	a1

`Change[Novel]` View

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

`Exchange[Novel]` View

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

Figure 4: Three views in global layer

Fig. 1. The `peer` attribute stores peer names. `Change[Novel]` is also a global view which unifies all `Change[Novel]` relations in a similar manner. `Exchange[Novel]` unifies all the underlying `From[Novel]` and `To[Novel]` relations in the local layer. Attributes `from_peer` and `to_peer` express the source and the destination of a record exchange, respectively. Attributes `from_id` and `to_id` contain the ids of the exchanged record in both peers.

Figure 5 illustrates the relationship between the local layer and the global layer.

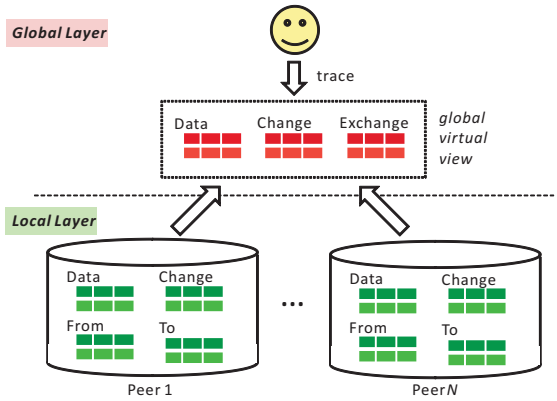

Figure 5: Local layer vs. global layer

4. TRACING QUERIES

In this section, we describe how to define tracing queries and introduce two execution modes.

4.1 Definition of Tracing Queries

When a tracing requirement occurs, the user needs to aggregate the related lineage information stored in the distributed peers. Since recursive processing in a P2P network is required to collect such information, the *datalog* language [2] appears to be one of the most promising language for specifying queries. Furthermore, by using datalog, we can reduce the program size greatly and can cope with various types of tracing queries [10, 12]. The language datalog¬ is defined by allowing negated literals in the bodies of rules [2]. In the following, we simply call it *datalog*. We introduce the notation and semantics of a tracing query with an example.

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 Q1 fulfills the requirement:*

Query Q1

```
BReach(P, I1) ← Data[Novel]('A', I2, 't1', 'a1'),
                Exchange[Novel](P, 'A', I1, I2, _)
BReach(P1, I1) ← BReach(P2, I2),
                 Exchange[Novel](P1, P2, I1, I2, _)
Origin(P) ← BReach(P, I),
            ¬ Exchange[Novel](_, P, _, I, _)
Query(P) ← Origin(P)
```

`P` *and* `I1` *are variables and '_' indicates an anonymous variable. Relation* `BReach` *defined by the first two rules means "Backward Reachable". It recursively traverses the arriving path of tuple* `(t1, a1)` *until it reaches the origin. The third 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 represents the final result expected by the user.* □

Note that the query is written using the three views in the global layer. The user does not need to consider how the actual data is distributed among the peers. The next is another example.

EXAMPLE 2. *Consider the following query Q2:*

Query Q2

```
Reach(P, I1) ← Data[Novel]('A', I2, 't1', 'a1'),
               Exchange[Novel]('A', P, I2, I1, _)
Reach(P, I1) ← Reach(P1, I2),
               Exchange[Novel](P1, P, I2, I1, _)
End(P) ← Reach(P, I),
         ¬ Exchange[Novel](P, _, I, _, _)
Query(P) ← End(P)
```

This query is similar to query Q1, except for exchanging `BReach` *and* `Origin` *by* `Reach` *and* `End`*. It finds the peers which located at the end of the peers that recursively copied* `(t1, a1)` *from peer A. In contrast to query Q1, its query result may change as time passes. For example, if we apply the query to our sample database, we get the result* `Query` $= \{(\texttt{C})\}$*. Assume that peer E copied the record from peer C. After that, the result will change as* `Query` $= \{(\texttt{E})\}$*.* □

As shown in these examples, datalog is so flexible that we can specify various queries for tracing. Other types of examples are shown in [10, 12].

4.2 Ad-hoc Queries and Continual Queries

Our framework supports two types of query execution modes, the *ad-hoc execution mode* and the *continual execution mode*. Depending on the execution mode, a query is called an *ad-hoc query* or a *continual query*, respectively.

First we consider ad-hoc queries. When query Q1 is issued from a user in an ad-hoc manner, the query is processed with the cooperation of distributed peers, then the result is returned the original query issuer (the initial peer). Ad-hoc queries are effective when we want to trace lineage information currently available in the network.

We can execute Q2 as an ad-hoc query, but there is a problem. Other peers may copy the target record after the query is executed. If we want to know up-to-date information, we need to issue ad-hoc queries repeatedly. To solve the problem, we introduce the *continual execution mode*. When a tracing query is executed in the continual execution mode, the query is firstly executed as if in the ad-hoc execution mode and an initial result is returned to the query peer, but the query is replicated in the related peers while the distributed query execution. The query is registered in each related peer as a *continual query*. A continual query monitors changes in its peer, and may report an incremental query result to the query peer when an additional result is obtained. A continual query may be copied repeatedly when some related events happen. To quit a continual query, a user explicitly removes the query to clear the states in the related peers.

Note that the continual execution mode is not effective for the queries asking past information only. For example, if we run query Q1 in the continual mode, the new result will not appear because the query only refers to past histories.

5. QUERY PROCESSING

The complete algorithms are described in [13]. In this section, we only describe the overview of query processing.

5.1 Query Mapping

Remember that tracing queries are described in datalog in terms of three global virtual views in the global layer. In order to process a tracing query, first we need to transform the query for distributed execution using the information in the local layer. The transformation is based on *mapping rules* [13]. The following shows an example of mapping.

EXAMPLE 3. *Query Q1 is mapped as follows:*

Mapped Query Q1

```
BReach(P, I1) ← Data[Novel]@'A'(I2, 't1', 'a1'),
                From[Novel]@'A'(I2, P, I1, _)
BReach(P1, I1) ← BReach(P2, I2),
                 From[Novel]@P2(I2, P1, I1, _)
Origin(P) ← BReach(P, I),
            ¬ From[Novel]@P(I, _, _, _)
Query(P) ← Origin(P)
```

We call the symbol `@` *a* location specifier*. If a constant peer name follows this symbol as* `@'A'`*, it means that the relation is located at peer A.* `From[Novel]@P2` *represents* `From[Novel]` *relation at peer* `P2`*, where* `P2` *is a variable representing a peer name. It is instantiated while the query processing.* □

Note that the mapped query is described by using relations in the local layer. In the global to local mapping step, we select one of the *executable* programs after applying the mapping rules. After that, we go to the actual query execution process.

5.2 Ad-hoc Query Processing

In the case of ad-hoc execution mode, to execute a mapped query, we further need to translate the query into the form that is suit for distributed execution. We employ the *seminaive method* [2], which is the most basic method for datalog query evaluation. The outline is given as follows:

1. First we derive query fragments for each *idb* (intensional database) based on the seminaive method.
2. Given query fragments (and intermediate relations, if the peer is not an initial peer), peer p performs query processing locally as possible. Using the terminology of deductive databases, we execute the query fragments until we reach the local *fixpoint*.

3. If the remaining part of the query process can be executed by other peers, p forwards the query fragments and intermediate relations to such peers $p_1, \ldots, p_n$. Peers $p_1, \ldots, p_n$ perform similar processes recursively.

4. Peer p merges the query results from $p_1, \ldots, p_n$ and own result then return the merged result. If p is called recursively from another peer, the result is returned to the peer. If p has no peer for forwarding in Step 2, it returns the own result only. After that, p deletes all the local intermediate data.

5. When the initial peer receives all the results, it returns the final result to the user by merging them.

EXAMPLE 4. *Figure 6 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. At first, the initial peer A executes the query locally and gets intermediate results* $\Delta^{\text{new}}_{\text{BReach}}$ *and* $\texttt{BReach}^{\text{new}}$. $\texttt{BReach}^{\text{new}}$ *contains the information of the peers which are on the path from peer A to the origin.* $\Delta^{\text{new}}_{\text{BReach}}$ *contains tuples which are new in the step. It drives the query process based on the seminative method. Since peer A have 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 is made by considering the contents of* $\Delta^{\text{new}}_{\text{BReach}}$.

After receiving the intermediate results from peer A, peer B starts the local query process and gets new $\Delta^{\text{new}}_{\text{BReach}}$ *and* $\texttt{BReach}^{\text{new}}$. *Based on the similar decision, the query is finally forwarded to peer D. In the query process of peer D, the third rule is executed because peer D is the origin. In this case, the seminaive query evaluation iterates twice in peer D and reaches the fixpoint. Since there are no following peers, we terminate the process, and then the results are returned in the backward direction through the forwarding path.* □

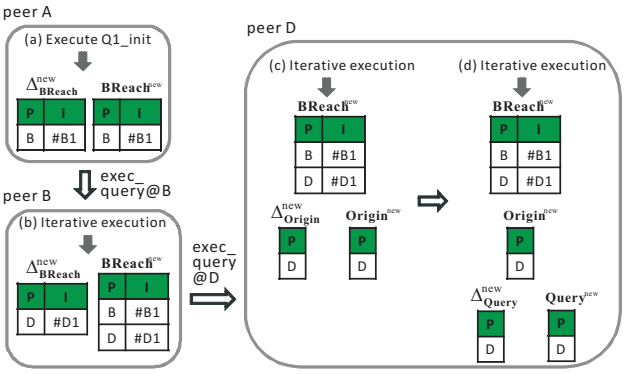

Figure 6: Execution of Q1 in ad-hoc mode

The seminaive method is quite popular for the query evaluation in deductive databases and also used in *declarative networking* [14]. The main difference between declarative networking and our approach is in their targets. In declarative networking, the target is to monitor information from a network such as a sensor network. Each node has connection information of neighboring nodes, but does not have network-wide information. Thus, recursive query processing is effective for traversing the connection paths. In contrast, a tracing query in our approach traverses the record exchange paths between peers. In this sense, its query processing is *content-based*; a query is forwarded to other peers if they have record exchange histories and the forwarding is not depend on the physical P2P network. In addition, our approach has two abstraction layers, the user layer and the global layer. Especially, the global layer is quite important to write a tracing query in an intuitive manner.

Although we only mentioned the use of the seminaive method, we performed some experiments for comparing the seminaive method and the *magic set method* in [11]. The result indicates that the magic set method is effective in some specific conditions, but the seminaive method is simple and applicable to any queries.

5.3 Continual Query Processing

As described in Section 4.2, a continual query is executed for a potentially long period of time, and is particularly useful in our context where information of record exchange is updated frequently in a distributed P2P network. Consider Example 2 again. The result of a tracing query may change when an update is performed in any related peers. To monitor updates, we introduce the *continual execution mode*.

A continual query can be executed based on the seminaive method as in the case of ad-hoc queries. The difference is that we store the partial results of a continual query as the *state* of the query. When an update occurs, we restart the query processing.

EXAMPLE 5. *We explain how query Q2 is executed as a continual query in an intuitive manner. First, the query is executed like an ad-hoc query. Figure 7 shows the result for our example. The result* $\texttt{Query} = \{(\texttt{C})\}$ *obtained in peer C is returned to peer A, then it is returned to the user as a final result. The execution is same as the ad-hoc mode, but the difference is that we do not delete the given queries and their intermediate data in the peers.*

Suppose that peer E copied the record `(t1, a1)` *from peer C now, and assume that its record id in peer E be* `#E1`. *In this case, some tuples are inserted in the local databases of peer C and E. Especially, the insertion of a new tuple in* `To[Novel]@'C'` *triggers the new query process. We restart the query evaluation process in peer C and utilizes* $\Delta^{\text{new}}_{\text{Reach}}$ *and* $\texttt{Reach}^{\text{new}}$, *which are stored in peer C as the state to contine the query. By reevaluating the query from peer C, we get new results as shown in Fig. 8. Based on the reevaluation, we obtain a new result* $\texttt{Query} = \{(\texttt{E})\}$. *This new result replaces the former result in peer A.* □

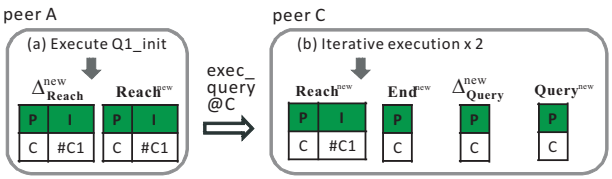

Figure 7: Continual query execution: before update

6. FRAMEWORK ENHANCEMENT BASED ON MATERIALIZED VIEWS

We describe on-going work for enhancing our framework based on materialized views. *Materialized views* play important roles in databases [8], especially for reducing query

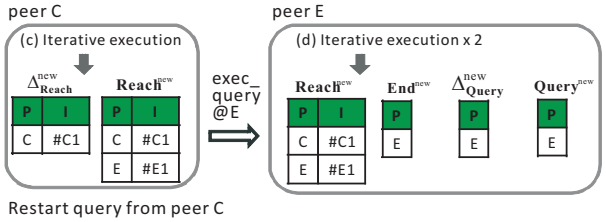

Figure 8: Continual query execution: after update

processing cost. In our context, materialized views are quite helpful because most of tracing queries (except for continual ones) use past histories. As described below, they are also useful for fault-tolerance because duplicated information is maintained in different peers.

In the following, we describe the basic idea to use materialized views effectively in our framework. In addition to own record exchange history, each peer maintains exchange histories which are reachable within k *hops* in the record exchange paths. For example, `From[Novel]@'A'` in Fig. 2 contains the information that record `#A1` is a copy of record `#B1` in peer B. If k is specified as $k = 2$, peer A also maintains the information that record `#B1` in peer B is a copy of record `#D1` in peer D. We can define similar replication methods for `Data` and `Change` relations. Datalog-based query definitions are useful for describing such replication schemes as materialized view definitions.

One benefit of the approach is that we can skip some peers when we traverse the record exchange path. For example, an execution of query Q1 can directly go from peer A to peer D without accessing peer B. Other benefit is in its use for *fault-tolerance*. Even if a peer is disconnected from the network due to some failure, we may be able to *recover* its historical information by combining the replicated information in the related peers.

The negative aspect is the increase of the maintainance cost not only in terms of storage cost but also in terms of processing cost. For example, when we maintain materialized view for `From[Novel]` with $k = 2$ as above, we need to perform transaction processing involving three peers when an update happens. Therefore, the tradeoff between the cost and the benefit would become an important issue. Currently, we are developing the maintainance method of materialized views and their effective use in query processing.

7. CONCLUSIONS AND FUTURE WORK

In my work, the concept of a traceable P2P record exchange framework was proposed [10, 12]. It is a unique approach to information exchange in a P2P network that incorporates the notion of lineage tracing. One of the important features of the framework is to maintain historical information in distributed peers and to integrate the information based on the "pay-as-you-go" approach [9]. The use of datalog [2] is another feature of the framework. A tracing process basically requires a recursive traversal along the path of record exchange. We can write various types of tracing queries in a compact manner using datalog. In addition, we already developed the query processing algorithms for both of the ad-hoc execution mode and the continual execution mode [13].

Considering the issues of performance improvement and fault-tolerance, we are now developing an enhanced framework with the support of materialized views. In addition, we are currently developing a prototype system based on our framework. The development will have positive feedbacks to improve our fundamental framework.

8. ACKNOWLEDGMENTS

This research was partly supported by the Grant-in-Aid for Scientific Research (#21013023, #22300034) from JSPS.

9. REFERENCES

[1] K. Aberer and P. Cudre-Mauroux. Semantic overlay networks. In *VLDB*, 2005. (tutorial notes).

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[4] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *Proc. VLDB*, pages 900–911, 2004.

[5] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *Proc. ACM PODS*, pages 1–12, 2008.

[6] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2):179–227, 2000.

[7] 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*, pages 1131–1133, 2007.

[8] A. Gupta and I. S. Mumick, editors. *Materialized Views*. MIT Press, 1999.

[9] A. Halevy, M. Franklin, and D. Maier. Principles of dataspace systems. In *Proc. ACM PODS*, pages 1–9, 2006.

[10] F. Li, T. Iida, and Y. Ishikawa. Traceable P2P record exchange: A database-oriented approach. *Frontiers of Computer Science in China*, 2(3):257–267, 2008.

[11] F. Li, T. Iida, and Y. Ishikawa. 'Pay-as-you-go' processing for tracing queries in a P2P record exchange system. In *Proc. DASFAA*, pages 323–327, 2009.

[12] F. Li and Y. Ishikawa. Traceable P2P record exchange based on database technologies. In *Proc. APWeb*, pages 475–486, 2008.

[13] F. Li and Y. Ishikawa. Query processing in a traceable P2P record exchange framework. *IEICE Transactions on Information and Systems*, E93-D(6):1433–1446, 2010.

[14] 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*, pages 97–108, 2006.

[15] W.-C. Tan. Research problems in data provenance. *IEEE Data Engineering Bulletin*, 27(4):45–52, 2004.

[16] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. CIDR*, pages 262–276, 2005.