PAPER Special Section on Info-Plosion

Query Processing in a Traceable P2P Record Exchange Framework

Fengrong LI[†a)], ***Student Member and*** **Yoshiharu ISHIKAWA**[††b)], ***Member***

SUMMARY As the spread of high-speed networks and the development of network technologies, *P2P technologies* are actively used today for information exchange in the network. While information exchange in a P2P network is quite flexible, there is an important problem—lack of reliability. Since we cannot know the details of how the data was obtained, it is hard to fully rely on it. To ensure the reliability of exchanged data, we have proposed the framework of a *traceable P2P record exchange* based on database technologies. In this framework, *records* are exchanged among autonomous peers, and each peer stores its exchange and modification histories in it. The framework supports the function of *tracing queries* to query the details of the obtained data. A tracing query is described in *datalog* and executed as a recursive query in the P2P network. In this paper, we focus on the query processing strategies for the framework. We consider two types of queries, *ad hoc queries* and *continual queries*, and present the query processing strategies for their executions.

key words: *information exchange, peer-to-peer networks, traceability, data provenance, query processing*

1. Introduction

In recent years, *peer-to-peer* (*P2P*) *technologies* have attracted much attention from both academic communities and industries [3]. A P2P network consists of a large number of autonomous *peers*. Peers cooperate to provide various information services such as information sharing and delivery without centralized control.

From the early stage of the P2P technologies, one of their main application fields was information exchange among peers. For example, Gnutella [12] is a well-known software system for P2P file exchange. P2P technologies were successful for flexible information exchange among peers in all over the world, but there exists an important issue—*lack of reliability*. By the term *reliability*, we mean the trustiness of the data exchanged in a P2P network. For example, when we get some data from a P2P network, how can we believe that the data is fully trustful?

In a typical P2P file exchange service, we cannot know details of the obtained data such as who created the data and which peers participated the process of a file exchange. It means that such systems are not *traceable*. The support of *traceability* is quite important for a reliable information exchange.

The traceability facility is also highly interested in the field of database research. The keyword *data provenance* (or *lineage*) is often used for describing the related concept [7], [8], [29]. In short, data provenance tries to give us evidences how a data item was obtained from other data items and why a data item exists in the database. So far, practical and theoretical methodologies for describing, querying, and maintaining provenance information have been proposed. Some projects focus on the data provenance issues in P2P information integration [18].

With the above background, we have proposed a framework for *traceable P2P record exchange* [20], [23]. We have extended the notion of data provenance to information exchange in a P2P network. In the framework, a *record* means a tuple-structured data item that obeys a predefined schema globally shared in a P2P network. 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 the database technologies. 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 "*pay-as-you-go*" approach [17] for tracing. We assume that tracing queries do not occur frequently so that it is not a wise idea to pay high maintenance cost only for the efficient 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 query processing issues in our traceable P2P record exchange framework. Although the basic concept of the framework was described in our former papers [20], [23], we did not show concrete procedures to evaluate tracing queries. In addition, this paper introduces a new type of queries called *continual queries* [24] in our framework. In contrast, the existing type of queries is called *ad-hoc queries*, which represents queries issued by users in an instantaneous manner. A continual query is stored in the related peers in a P2P network and used to monitor the specified events to occur. We show that continual queries are also quite useful to represent some types of traceability requirements.

One example of the application fields that our P2P record exchange framework assumes is information ex-

Manuscript received September 4, 2009.
Manuscript revised January 4, 2010.

[†]The author is with the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8601 Japan.
[††]The author is with the Information Technology Center, Nagoya University, Nagoya-shi, 464–8601 Japan.
a) E-mail: lifr@db.itc.nagoya-u.ac.jp
b) E-mail: ishikawa@itc.nagoya-u.ac.jp

DOI: 10.1587/transinf.E93.D.1433

change in scientific communities. For example, large databases are maintained distributedly in the bioinformatics area, but there is a big problem of data provenance [7], [29]. Copied data is managed independently in each organization and researchers often modify and annotate the local copied data, and to make matters worse, other organization may copy the modified data. We should esteem the autonomy of each organization, but the original source and modification histories should be able to be traceable. Another example of the application field is cooperative distributed digital libraries. In the research field of digital libraries, P2P technologies attract researchers to enable next-generation digital libraries. For example, BRICKS [6], organized as an EU IST Project, developed a framework of P2P-based distributed digital libraries. In their framework, digital libraries and museums can participate cooperative resource sharing as peers.

Based on the above reasons, we aim at flexible P2P resource sharing with esteeming autonomity of each peer. Each peer in our framework is independent and can exchange information with other peers. We estimate that each peer is a trusted organization such as a research center and a digital library so that privacy and anonymity issues are out of scope. In addition, failure and unexpected leave of a peer is not a critical problem compared to information sharing among untrusted peers.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives the overview of the traceable P2P record exchange system. Section 4 defines tracing queries and presents their two execution modes. Section 5 and Sect. 6 provide the query processing strategies for ad-hoc queries and continual queries, respectively. Section 7 discusses the issues and Sect. 8 concludes the paper and addresses the future work.

2. Related Work

There are a variety of research topics regarding 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) are loosely but cooperatively shared in the network. In this sense, a highly related project is ORCHESTRA [13], [19], [27], 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. In contrast to the conventional approaches of information integration, we do not consider schema heterogeneity to simplify the problem; a record schema is globally shared in a network.

A related concept is a *dataspace system* [17]. Roughly speaking, it is a topic of information integration, but focuses on a more 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 [17]. Since we can assume tracing queries (integration of histories) 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.

Another related topic is *data provenance*. It tries to give users some evidence about a data item, for example, why the item is in the database, how it is obtained from other data sources, etc. Historical information to support data provenance is often called *lineage*. The target field of data provenance is quite wide and it covers data warehousing [10], [11], uncertain data management [4], [30], database curation [7], and other scientific fields such as bioinformatics [5]. 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 of data provenance, 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. ORCHESTRA, a P2P information integration system [13], [19], [27], has a feature of data provenance, but it is limited in the context of data integration. Our approach mainly focuses on the data provenance issue in P2P record exchange.

A tracing query in our framework is executed as a recursive query executed in a P2P network. It recursively traverses the peers which contain related historical information. To describe a tracing query in a user-friendly manner, we use the *datalog* language, which is a well-known query language for deductive databases [2]. Query processing based on the deductive database approach was not a hot topic in recent years, but the situation is now changing. As proved in the *declarative networking* project [9], [26], [28], a declarative recursive query is a very powerful tool for network-oriented database applications such as P2P and sensor data aggregation. We can find several related ideas for executing and optimizing declarative queries in a network [16], [25], [32].

The idea of a *continual query* [24] is also related with our research. When a continual query is given to a query processing system, it is registered in the system and executed continually for the incoming data items. Continual queries are used to describe requirements to cope with sensor data, internet information delivery, etc. In our framework, a continual query is used to represent a tracing requirement to monitor changes happened in the P2P network.

3. System Framework

In this section, we describe the overview of the traceable

P2P record exchange framework. It is based on our former work [20], [23], but the terminology is revised. Since our main concern in this paper is the query processing issue, we omit some topics such as record search, registration and deletion and failure handling. Please refer to [20], [23] for their details.

3.1 Traceable P2P Record Exchange

As an example, let us assume that information about novels is shared among peers in a P2P network. Figure 1 shows an example record set `Novel` owned by a peer that 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 the same. In our traceable record exchange framework, peers can behave autonomously and exchange records when required. A peer can search records managed in other peers in a P2P network by specifying search conditions, and the peer can select and register the retrieved records into its *record management system*. After that, the records are under the management of the peer until explicit deletion instruction issued by the peer.

A *traceability problem* occurs, for instance, when the peer wishes to ask the following question: "Where did the record `(Pride and Prejudice, Jane Austen, English, 1813)` come from?" To support such a question, we need to provide the *traceability* facility in our framework. It implies that we should store all the record exchange histories in a P2P network and we need to provide a query language to express lineage queries.

To provide a traceability facility in our framework, we take the following principles:

- All the information required for tracing is maintained in distributed peers. Each peer maintains its own historical information, which consists of creation (registration), modification, deletion, and exchange histories, related to the peer itself.
- When a tracing query is issued, the query is processed by coordinating related peers in the P2P network. Historical information stored in related peers is collected for answering the query.
- To help users to write queries, an abstraction layer, called the *global layer*, is provided. In the global layer, all the data in a P2P network is integrated in global virtual views. In contrast, the underlying layer mentioned above is called the *local layer*. User queries are written by the *datalog* query language [2].

The underlying idea behind these design principles is the notion of "pay-as-you-go" data integration [17]. Since copies and updates are performed everywhere in a distributed autonomous P2P network, it is costly to maintain the historical information in a central server or several hub servers. Instead of that, each peer in our framework only maintains minimum historical information related to the peer. It means that we need to aggregate the required historical information from the corresponding peers when a tracing query is issued from a user—the user should pay the cost when he traces information. While the tracing cost is high, the principles are reasonable due to the following reasons:

- Since tracing queries are not issued quite often, the "pay-as-you-go" approach contributes to the low maintenance cost.
- A peer may have limited interests and exchange records within a group of peers which shares the local interests. In this case, a tracing query is only related with the peers in the group. Thus, it is not a wise idea to collect all the historical information into a centralized server.

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

Fig. 1 Example record set `Novel`.

3.2 Three Layer Model

To represent records and their historical information, we employ the layered architecture with different abstraction levels. It is called the *three layer model*.

3.2.1 User Layer

The *user layer* directly supports what users see. Figure 2 shows a simplified example with four peers. Each peer maintains a `Novel` record set that has two attributes `title` and `author`.

In our framework, every peer can act as a provider and a searcher. A peer can find desired records from other peers by issuing a query. In addition, a peer can register the retrieved or created records into the local record management system, and can modify and delete records in the local system. For example, the record `(t1, a1)` in peer A in Fig. 2 may have been copied from peer B and registered in peer A's record management system.

3.2.2 Local Layer

The *local layer* is used for representing each peer's own

Peer A

title	author
t1	a1
t3	a3

Peer B

title	author
t1	a1
t4	a4

Peer C

title	author
t1	a1

Peer D

title	author
t1	a1

Fig. 2 Record sets in four peers.

id	*title*	*author*
#A1	t1	a1
#A2	t2	a2
#A3	t2	a3
#A4	t3	a3
#A5	t5	a5

Fig. 3 `Data[Novel]@'A'`.

from_id	to_id	time
–	#A2	...
#A2	–	...
#A2	#A3	...
#A3	–	...
#A3	#A4	...
#A5	–	...

Fig. 4 `Change[Novel]@'A'`.

id	from_peer	from_id	time
#A1	B	#B1	...
#A5	C	#C2	...

Fig. 5 `From[Novel]@'A'`.

id	to_peer	to_id	time
#A1	C	#C1	...
#A5	B	#B3	...

Fig. 6 `To[Novel]@'A'`.

record set with local historical information using four relations. For example, peer A shown in Fig. 2 contains the following relations:

- `Data[Novel]@'A'`: It maintains records owned by peer A. Figure 3 shows the example. Every record has its own record id for the maintenance purpose. Each record id should be unique in the entire P2P network. Note that there are additional records compared to Fig. 2. They are *deleted* records and normally hidden from the user.
  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. We also allow a variable follows a location specifier such as `@P`.
- `Change[Novel]@'A'`: It is used to hold record creation (or registration), modification, and deletion histories. Figure 4 shows its example for peer A. Attributes `from_id` and `to_id` express the record ids before/after a modification. Attribute `time` represents the timestamp. When the value of the `from_id` attribute is the null value (−), it represents that the record was created at the peer. Similarly, when the value of the `to_id` attribute is the null value, it means that the record was deleted locally. Although we simply modeled data updates as insertions of tuples, real applications may have additional semantics such as keys and dependencies, and update requirements may be different in applications. We would like to consider this problem in the future work.
- `From[Novel]@'A'`: 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. Attribute `time` stores the timestamp information. The first tuple in Fig. 5 shows the record with id `#A1` is a copy of the record with id `#B1` at peer B.
- `To[Novel]@'A'`: It plays an opposite role of `From[Novel]@'A'` and stores information which records were sent from peer A to other peers. Figure 6 shows its example.

Note that `From[Novel]` and `To[Novel]` contain duplicates, which are stored in different peers. For example, for the tuple of `From[Novel]@'A'` in Fig. 5, 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.

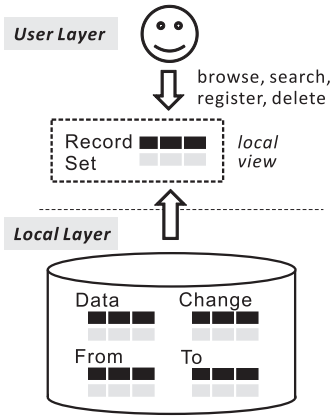


Fig. 7 Local layer vs. user layer.

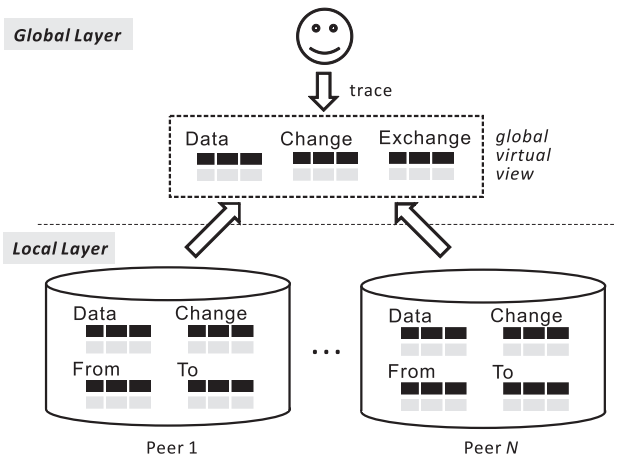


Fig. 8 Local layer vs. global layer.

The relationship between the local layer and the user layer is shown in Fig. 7. A record set in the user layer corresponds to a virtual restricted view constructed from the Data relation in the underlying local layer.

3.2.3 Global Layer

The *global layer* provides virtual integrated views of all information in the P2P network. Its main role is to support users for writing tracing queries and users do not need to consider how historical information is distributed in the network. Figure 8 illustrates the relationship between the local layer and the global layer. In the global layer, three relational views are constructed by unifying all the relations in peers. Relation `Data[Novel]` in Fig. 9 expresses a view that unifies all the `Data[Novel]` relations in peers A to

peer	id	*title*	*author*
A	#A1	t1	a1
A	#A2	t2	a2
A	#A3	t2	a3
A	#A4	t3	a3
A	#A5	t5	a5
B	#B1	t1	a1
B	#B2	t4	a4
B	#B3	t5	a5
C	#C1	t1	a1
C	#C2	t5	a5
D	#D1	t1	a1

Fig. 9 View `Data[Novel]`.

peer	from_id	to_id	time
A	–	#A2	...
A	#A2	–	...
A	#A2	#A3	...
B	–	#B2	...
B	#B3	–	...
C	–	#C2	...
C	#C2	–	...
D	–	#D1	...

Fig. 10 View `Change[Novel]`.

from_peer	to_peer	from_id	to_id	time
B	A	#B1	#A1	...
A	C	#A1	#C1	...
D	B	#D1	#B1	...
C	A	#C2	#A5	...
A	B	#A5	#B3	...

Fig. 11 View `Exchange[Novel]`.

D shown in Fig. 2. The `peer` attribute stores peer names[†]. Relation `Change[Novel]` shown in Fig. 10 is also a global view which unifies all `Change[Novel]` relations in a similar manner.

`Exchange[Novel]` shown in Fig. 11 unifies all the underlying `From[Novel]` and `To[Novel]` relations in a global view. Attributes `from_peer` and `to_peer` express the origin and the destination of record exchanges, respectively. Attributes `from_id` and `to_id` contain the logical ids of the exchanged records in both peers.

In summary, the local layer manages the records and historical information for each peer. The user layer is a view on the local layer, and hides lineage information from the user. It is limited for a specific peer. The global layer is also located over the local layer, but it provides a global virtual view by integrating all the information in distributed peers. The global views in Figs. 9, 10, and 11 are virtual views which unify all of the related information in the P2P network. Although they are used for writing tracing queries, they are not materialized.

4. Tracing Queries

In this section, we describe how tracing queries are defined 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* query language [2] is used in our framework. Because if we want to trace the provenance of data, recursive query is needed. Since the lineage information in our framework has a recursive nature, datalog 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 [20], [23]. First 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)
```

We introduce some terms used for describing a datalog program [2]. A predicate corresponding to a stored relation, such as `Data[Novel]`, is called an *edb* (extensional database) predicate. An *edb* predicate can only appear in rule bodies. While a predicate representing a derived relation that does not exist in the database, such as `BReach`, is called an *idb* (intentional database) predicate. `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. □

A tracing query is defined as a *datalog*$^{\neg}$ program with some restrictions. The language datalog$^{\neg}$ is defined by allowing negated literals in the bodies of rules [2]. In the following, we simply call it *datalog*.

Definition 1 (Tracing Query): A *tracing query* Q is a *datalog*$^{\neg}$ *query* that satisfies the following conditions:

1. Q only contains Data, Change, and Exchange relations as *edb* predicates.
2. The arities of Data predicates appeared in Q should consistent with the definition of the target record schema.
3. Q is a *safe* datalog query.
4. Q is a *linear* datalog query.
5. Q is *stratifiable*.
6. All recursions appeared in Q should be *self-recursions* over Exchange relations.

[†]In our framework, peer name is an overlay identifier in the overlay network. We assume that we can send a message to a peer if we know its peer name.

7. The body of each rule in Q does not contain different location specifier variables. □

The conditions 1 to 7 have two aims: (a) to set the scope and the structure of a tracing query (1 to 3) and (b) to limit the expressive power of a datalog query just enough for representing a tracing query to simplify the query processing (4 to 7). Condition 1 is to ensure that all of the tracing queries should only use the three relations as *edb* predicates. Condition 2 is for consistency with the target record structure. For example, the `Data[Novel]` predicate in query Q1 satisfies the condition since its arity is four. Condition 3 is required for guaranteeing the ranges of all variables are restricted [2]—a datalog query is said *safe* if it satisfies some reasonable syntactic conditions [31]. It is a sufficient condition that a datalog query does not have the problems of domain dependence and infiniteness of answers. Condition 4 is useful for simplifying query processing. A datalog rule with head relation R is *linear* if there is at most one atom in the body of the rule whose predicate is mutually recursive with R. A datalog program is *linear* if each rule of the program is linear [2]†. Fortunately, we do not need mutual recursions for tracing. In addition, nonlinear recursions can frequently be made linear by rewriting. A linear datalog has a benefit that it can be evaluated by a simple algorithm compared to a non-linear datalog query [2]. Condition 5 is required for efficient query processing while allowing a reasonable expressive power including negation. Our tracing query can be evaluated in polynomial time in terms of the database size because datalog with stratified negation is in DB-PTIME [31]. Condition 6 is a realistic restriction for our tracing problem. As shown in query Q1, recursion in our framework is only used for traversing Exchange relations. Such a query can be described using self-recursion. The restriction greatly simplifies query processing because we do not need to consider general recursions. Condition 7 ensures that we can perform distributed query execution by query forwarding. Its detail is mentioned in Sect. 5. If we allow different location specifier variables in a rule body, it may result in arbitrary distributed joins between peers. That is highly costly and is not used when we issue a query for a tracing purpose.

Note that this restriction is not enough for a valid query in terms of distributed query execution. We introduce an additional condition in Sect. 5.

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. In this section, we describe how tracing queries are defined and introduce two execution modes.

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). In this sense, such a query is called an *ad-hoc query*. We say that an ad-hoc query is executed in the *ad-hoc execution mode*. As shown in this example, datalog is so flexible that we can specify various queries for tracing.

Ad-hoc queries are effective when we want to trace lineage information currently available in the network. However, the facility is not suffice for supporting some types of tracing requirements. We illustrate the problem by an 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, _)
Query(P) ← Reach(P, _)
```

The query returns all the peers that copied the target record `(t1, a1)` in peer A. 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 reports 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.

Example 2 (Continued): If we execute query Q2 for our example databases, we get an initial result tuple `(C)`, which is a peer that copied the record `(t1, a1)` from peer A. While the execution, the query is copied into peer C as a continual query. The continual query is triggered, for instance, when peer E copies the record `(t1, a1)` from peer C. The query derives an incremental result tuple `(E)`, and it is sent to the query peer A. The process enables continual tracing in a P2P network. □

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.

†The following is an example of a non-linear program because two R's appear in the body of the second rule.

```
R(I1, I2) ← Exchange[Novel](_, 'A', I1, I2, _)
R(I1, I2) ← R(I1, I3), R(I3, I2)
```

5. Ad-Hoc Query Processing

In this section, we describe the query processing method for ad-hoc tracing queries. Some of the techniques described in this section are reused for continual tracing queries discussed in the next section.

5.1 Query Mapping

Remember that tracing queries are described in datalog in terms of 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 mapping rules are summarized in Fig. 12, where R represents the record name and *attrs* means its attribute list. Rules r_1 and r_2 are straightforward. Rules r_3 and r_4 show two alternatives for rewriting Exchange relation. We explain how to select an appropriate one later.

We show 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)
```

`From[Novel]@P2` represents `From[Novel]` relation at peer `P2`, where `P2` is a variable representing a peer name. The variable is instantiated while the query processing. Note that the mapped query only accesses relations in the local layer. □

Note that an application of the rules in Fig. 12 may produce multiple mapped queries. They may contain queries which are not efficiently evaluable. In the following, we describe how to decide a given mapped query is executable. First, we define the notion of an *accessible variable*.

Definition 2 (Accessible Variable): A variable X in a rule body of a mapped query is *accessible* if either of the following conditions is satisfied:

1. X appears in a positive (non-negated) *idb* predicate in the body.
2. X appears in a positive *edb* predicate in the body, and the location specifier of the *edb* predicate is bound to a constant or an accessible variable.
3. There exists an equality literal $X = Y$ in the body, where Y is an accessible variable. □

r_1: $\mathrm{Data}[R](\mathrm{peer}, \mathrm{id}, \mathit{attrs}) \Rightarrow \mathrm{Data}[R]@\mathrm{peer}(\mathrm{id}, \mathit{attrs})$
r_2: $\mathrm{Change}[R](\mathrm{peer}, \mathrm{from\_id}, \mathrm{to\_id}, \mathrm{time})$
$\Rightarrow \mathrm{Change}[R]@\mathrm{peer}(\mathrm{from\_id}, \mathrm{to\_id}, \mathrm{time})$
r_3: $\mathrm{Exchange}[R](\mathrm{from\_peer}, \mathrm{to\_peer}, \mathrm{from\_id}, \mathrm{to\_id}, \mathrm{time})$
$\Rightarrow \mathrm{To}[R]@\mathrm{from\_peer}(\mathrm{from\_id}, \mathrm{to\_peer}, \mathrm{to\_id}, \mathrm{time})$
r_4: $\mathrm{Exchange}[R](\mathrm{from\_peer}, \mathrm{to\_peer}, \mathrm{from\_id}, \mathrm{to\_id}, \mathrm{time})$
$\Rightarrow \mathrm{From}[R]@\mathrm{to\_peer}(\mathrm{to\_id}, \mathrm{from\_peer}, \mathrm{from\_id}, \mathrm{time})$

Fig. 12 Global to local mapping rules.

We illustrate the idea using mapped query Q1 as an example. In the first rule body, the appeared variables `I1`, `I2`, and `P` are all accessible because Condition 2 is satisfied. In the second rule body, `P2` and `I2` satisfy Condition 1, thus they are accessible. Since `P2` is accessible, Condition 2 says that `P1` and `I1` are accessible.

Using this notion, we define an executable query.

Definition 3 (Executable Query): A mapped query is *executable* if every location specifier that appears in the bodies of the query is bound to a constant peer name or an accessible variable. □

Let us consider an example of a non-executable mapped query. If we replace the first rule of mapped query Q1 with the following rule, we get an another candidate.

```
BReach(P, I1) ← Data[Novel]@'A'(I2, 't1', 'a1'),
                To[Novel]@P(I1, 'A', I2, _)
```

For the evaluation of this rule, we need to access `To[Novel]` relations in *all* the peers in the P2P network. It is quite inefficient or not impossible. By introducing the notion of an executable query, we can exclude such inefficient queries.

In summary, the global to local mapping step selects one of the executable programs after applying the mapping rules. If there are multiple executable candidates, we can select one of them arbitrarily. For the most of cases, however, we will have a unique candidate.

5.2 Deriving Query Fragments Based on Seminaive Method

To execute a mapped query, we further need to translate the query into the form that is suit for distributed execution. In this paper, we employ the *seminaive method* [2], which is the most basic method for datalog query evaluation. The seminaive method takes a *bottom-up* approach and its query process is performed based on *forward chaining*. As described later, forward-chaining can be directly applicable for query forwarding between distributed peers.

Given a mapped query Q, we can derive *query fragments* as follows:

1. For each rule $S(u) \leftarrow T_1(v_1), \ldots, T_n(v_n)$ in Q such that Q does not contain an *idb* predicate in the body, we create the following rule:
$$\boxed{\Delta_S^{\mathrm{new}}(u) \leftarrow T_1(v_1), \ldots, T_n(v_n)}$$
We call a program which is made by collecting all the transformed queries Q_init.
2. Next, we construct a program Q_S for each *idb* predicate S in Q. Let $S(u) \leftarrow T_1(v_1), \ldots, T_n(v_n)$ be a rule which defines S and contains an *idb* predicate in the

body, where $u, v_1, v_2, \ldots$, are lists of variables and constants. Since we are considering that a linear datalog program, the rule body has only one appearance of an *idb* predicate. Let the predicate be T_j. We construct a query fragment Q_S for *idb* predicate S as follows:

$$\begin{array}{l} temp_S(u) \leftarrow T_1(v_1), \ldots, \Delta_{T_j}(v_j), \ldots, T_n(v_n) \\ \Delta_S^{\text{new}} := temp_S - S \\ S^{\text{new}} := S \cup \Delta_S^{\text{new}} \end{array}$$

where ':=' denotes an assignment between relations.

We show an example of the derivation next.

Example 4: We derive query fragments from mapped query Q1. First, Q1_init is constructed by extracting rules that do not contain *idb* predicates in the bodies.

Q1_init

```
Δ^new_BReach(P, I1) ← Data[Novel]@'A'(I2, 't1', 'a1'),
                      From[Novel]@'A'(I2, P, I1, _)
```

Next, the query fragments are constructed for *idb* predicates `BReach`, `Origin`, and `Query`, respectively.

Q1_BReach

```
temp_BReach(P1, I1) ← Δ_BReach(P2, I2),
                      From[Novel]@P2(I2, P1, I1, _)
Δ^new_BReach := temp_BReach − BReach
BReach^new := BReach ∪ Δ^new_BReach
```

Q1_Origin

```
temp_Origin(P) ← Δ_BReach(P, I),
                 ¬ From[Novel]@P(I, _, _, _)
Δ^new_Origin := temp_Origin − Origin
Origin^new := Origin ∪ Δ^new_Origin
```

Q1_Query

```
temp_Query(P) ← Δ_Origin(P)
Δ^new_Query := temp_Query − Query
Query^new := Query ∪ Δ^new_Query
```

The three query fragments correspond to the iteration step of the seminaive method. □

5.3 Distributed Query Execution

5.3.1 Outline

We describe how query fragments are executed in a distributed P2P environment. The outline is given as follows:

1. 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*.
2. If the remaining part of the query process can be executed by other peers, p forwards the query fragments and intermediate relations to such peer $p_1, \ldots, p_n$. Peers $p_1, \ldots, p_n$ perform similar processes recursively. Peer p waits the query results from the peers.
3. 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 other 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.
4. When the initial peer receives all the results, it returns the final result to the user by merging them.

5.3.2 Query Execution Algorithm

We formulate the procedure in Algorithm 1. The algorithm receives query fragments Q_{init} and $\{Q_S\}$, where $\{Q_S\}$ is a set of query fragments for all *idb* predicates. Lines 1 to 5 correspond to the initialization step, where $\{S\}$ denotes the set of all *idb* predicates. The notations such as $\{S\}$ are not conventional, but used for simplifying the presentation. If we write $\{X\}$ in Algorithm 1, $\{X\}$ means a set of instances which made as instantiations of X. For example, $\{S\} = \{BReach, Origin, Query\}$ for Query 1 and $\{Q_S\}$ is a set of query fragments, each of which has the form defined in Sect. 5.2.

Lines 6 to 11 represent the local iteration step. We say that $\{Q_S\}$ is *locally executable* when there exists at least one $Q_S \in \{Q_S\}$ that satisfies the following condition: the value of $\Delta_{T_j}(v_j)$, appeared in the body of the rule $temp_S(u) \leftarrow T(v_1), \ldots, \Delta_{T_j}(v_j), \ldots, T_n(v_n)$ in Q_S, is not empty. We perform iterative execution while $\{Q_S\}$ is locally executable.

Lines 12 to 20 are called the query forwarding step. We say that $\{Q_S\}$ is *forwardable* when there exists at least one $Q_S \in \{Q_S\}$ that satisfies the following condition: the body of the rule $temp_S(u) \leftarrow T(v_1), \ldots, \Delta_{T_j}(v_j), \ldots, T_n(v_n)$ in Q_S contains a location specifier variable and it is bound

Algorithm 1 exec_query($Q_{\text{init}}, \{Q_S\}$)

```
Input: Q_init, {Q_S}: set of all Q_S
Output: {S^new}: query result
 1: // initialization
 2: Δ^new_S ← ∅, for each S ∈ {S}
 3:                                   ▷ {S} is a set of all idb predicates
 4: execute Q_init                    ▷ some Δ^new_S are updated
 5: S^new ← Δ^new_S, for each S ∈ {S}
 6: // iterative execution
 7: while {Q_S} is locally executable do
 8:     Δ_S ← Δ^new_S, for each S ∈ {S}
 9:     S ← S^new, for each S ∈ {S}
10:     execute Q_S, for each Q_S ∈ {Q_S}
11: end while
12: // query forwarding
13: if {Q_S} is fowardable then
14:     foreach target peer p do
15:         {S^res} ← exec_query@p({Q_S}, {Δ^new_S}, {S^new})
16:                   ▷ same algorithm except without initialization step
17:         S^new ← S^new ∪ S^res, for each S ∈ {S}
18:                                  ▷ merge the results of forwarded queries
19:     end for
20: end if
21: return {S^new}
```

to peer names[†] in $\Delta_{T_j}(v_j)$. Since our tracing query is safe, we can find assignments (peer names) for the location specifier variable. That means we should forward the query to the peers. The forwarding query includes the query fragments $\{Q_S\}$ and the current partial results $\{\Delta_S^{\text{new}}\}$ and $\{S^{\text{new}}\}$. The current peer waits the query results then merges them with the local result. Finally, $\{S^{\text{new}}\}$ is returned as the query result.

At line 15, we call other peers recursively. Each forwarded peer uses the similar query algorithm to perform the local query processing. The exception is that it receives arguments $\{Q_S\}$, $\{\Delta_S^{\text{new}}\}$, and $\{S^{\text{new}}\}$. Note that we used a for loop in lines 14 to 19 to simplify the presentation, but we can perform parallel query execution for efficiency: queries are first forwarded, then the peer waits the response by asynchronous communications.

Finally note that the algorithm returns $\{S^{\text{new}}\}$ at the end (line 21), but the initial peer only has to return the *idb* predicate `Query`, which the user actually wanted.

5.3.3 Example of Query Execution

We illustrate how query Q1 is processed using the database instances shown in Fig. 3 to Fig. 11. Figure 13 illustrates the query processing steps.

Example 5: Let us assume that query Q1 is issued at peer A. First, peer A executes Q1_init[††]. Since the location specifier constant in the rule body of Q1_init is `@A`, we can execute the initialization query locally[†††]. As the result of the initialization step, we get $\Delta_{\texttt{BReach}}^{\text{new}} = \texttt{BReach}^{\text{new}} = \{(\texttt{B}, \#\texttt{B1})\}$ as shown in Fig. 13 (a). Other relations are still empty.

Since the iteration step is not applicable to the current status, we go to the query forwarding step. We can notice that Q1_BReach and Q1_Origin satisfy the condition and $\Delta_{\texttt{BReach}}^{\text{new}}$ is not empty, thus $\{Q_S\}$ is forwardable. We forward the query to the target peer B, which has an entry in $\Delta_{\texttt{BReach}}^{\text{new}}$. Note that we need to forward to multiple peers if we have multiple entries in $\Delta_{\texttt{BReach}}^{\text{new}}$, but query Q1 asks about "origin" so that we have only one entry.

After receiving the forwarded query $\{Q_S\}$ and the partial result $\Delta_{\texttt{BReach}}^{\text{new}} = \texttt{BReach}^{\text{new}} = \{(\texttt{B}, \#\texttt{B1})\}$, peer B starts query execution. By executing Q1_BReach, we get $\Delta_{\texttt{BReach}}^{\text{new}} = \{(\texttt{D}, \#\texttt{D1})\}$ and $\texttt{BReach}^{\text{new}} = \{(\texttt{B}, \#\texttt{B1}), (\texttt{D}, \#\texttt{D1})\}$, as shown in Fig. 13 (b). Next, Q1_Origin is executed but no tuples are generated.

Then, peer B forwards the query to the next peer D. Peer D receives $\Delta_{\texttt{BReach}} = \{(\texttt{D}, \#\texttt{D1})\}$ and $\texttt{BReach} = \{(\texttt{B}, \#\texttt{B1}), (\texttt{D}, \#\texttt{D1})\}$. The execution of Q1_BReach results in $\Delta_{\texttt{BReach}}^{\text{new}} = \emptyset$ and $\texttt{BReach}^{\text{new}} = \{(\texttt{B}, \#\texttt{B1}), (\texttt{D}, \#\texttt{D1})\}$, and the execution result of Q1_Origin is $\Delta_{\texttt{Origin}}^{\text{new}} = \texttt{Origin}^{\text{new}} = \{(\texttt{D})\}$, as shown in Fig. 13 (c).

Peer D can perform one more iteration. By executing Q1_Query, we get $\Delta_{\texttt{Query}}^{\text{new}} = \texttt{Query}^{\text{new}} = \{(\texttt{D})\}$ as Fig. 13 (d). Since the query cannot continue further, peer D returns the result to peer B, and then peer B returns the result to peer A. Thus, the query finishes and we get $\texttt{Query}^{new} = \{\texttt{D}\}$. □

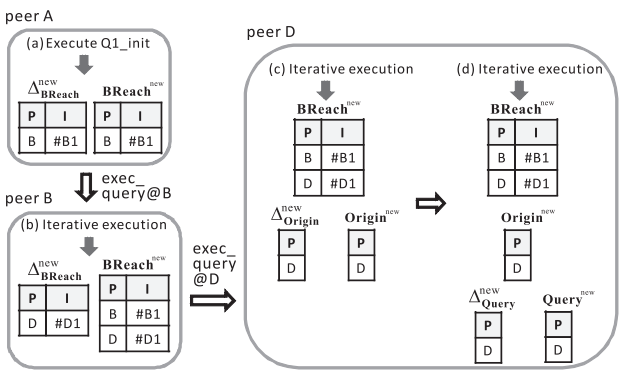


Fig. 13 Example of query execution.

5.4 Discussion

We describe some additional issues related to query processing.

5.4.1 Stratified Queries

Queries including negations may need to be evaluated by stratified evaluation based on multiple strata. We show an example here.

Example 6: Consider the following query Q4:

Query Q4

```
Reach1(P, I1) ← Data[Novel]('A', I2, 't1', 'a1'),
                Exchange[Novel]('A', P, I2, I1, _)
Reach1(P, I1) ← Reach1(P1, I2),
                Exchange[Novel](P1, P, I2, I1, _)
Reach2(P, I1) ← Data[Novel]('A', I2, 't2', 'a2'),
                Exchange[Novel]('A', P, I2, I1, _)
Reach2(P, I1) ← Reach2(P1, I2),
                Exchange[Novel](P1, P, I2, I1, _)
Query(P) ← Reach1(P, _), ¬ Reach2(P, _)
```

The query finds the peers such that they have direct or indirect copies of record `(t1, a1)` of peer A and that they do not have the copies of record `(t2, a2)` of peer A. Two *idb* predicates `Query` and `Reach2` belong to different strata because the body for `Query` includes `Reach2` in a negated form [2]. We can execute this query in two phases:

1. First we execute third and fourth rules for `Reach2` as the first stratum. We can use Algorithm 1 and we get an instance of `Reach2`.
2. As the second stratum, we execute the remaining rules. We treat `Reach2` as if a base relation.

□

[†]Strictly speaking, the peer names should be different from the current peer. If the peer names are same as the current peer, we consider that the query is locally executable.

[††]Note that Q1_init (and other queries) can be issued as a SQL query if the underlying RDBMS accepts SQL. Transformation of a conjunctive query to an SQL query is straightforward.

[†††]If the peer name is different from the current peer (e.g., `@B`), a remote query is issued to the peer.

As described before, we assume that given tracing queries are stratifiable. Therefore, we can execute all the tracing queries using one or more strata as shown in this example.

5.4.2 Query Optimization Based on Magic Set-Based Rewriting

We consider a query optimization issue based on the well-known magic set-based technique. Consider the following example.

Example 7: The following query detects whether peer C copied the record `(t1, a1)` owned by peer B:

Query Q5

```
Reach(P, I1) ← Data[Novel]('B', I2, 't1', 'a1'),
               Exchange[Novel]('B', P, I2, I1, _)
Reach(P1, I1) ← Reach(P2, I2),
                Exchange[Novel](P2, P1, I2, I1, _)
Query(I) ← Reach('C', I)
```

Of course, the query can be executed using Algorithm 1. However, the algorithm does not use the information of the constant 'C' that appears in the third rule. We have a possibility of efficient execution by "pushdown" the constraint of the constant in the early stage of query processing. □

As a strategy for efficient execution of datalog programs, a well-known approach for this problem is the *magic set method* [2]. By modifying a given program, it simulates "selection pushdown" for the top-down evaluation approach within the bottom-up evaluation approach. Once a program is modified by the magic set-based rewriting, we can execute the program using the seminaive method.

Example 7 (Continued): Query Q5 can be rewritten as a magic set-based program as follows:

Translated Query Q5

```
Reach(P, I1) ← magic_Reach(P, I1),
               Data[Novel]@'B'(I2, 't1', 'a1'),
               From[Novel]@P(I1, 'B', I2, _)
Reach(P1, I1) ← magic_Reach(P, I1), Reach(P1, I2),
                From[Novel]@P(I1, P1, I2, _)
magic_Reach(P1, I2) ← magic_Reach(P, I1),
                      From[Novel]@P(I1, P1, I2, _)
magic_Reach('C', I) ←
Query(I) ← Reach('C', I)
```

After the transformation, we can execute this program using the seminaive method in Algorithm 1. Assume that this query is executed at peer B. In this case, the body of the fourth rule above is empty. It is used to define the tuples of the `magic_Reach` relation, and it works as an initial goal for the magic set evaluation process. In other words, the fourth rule defines the actual starting point; it first triggers the evaluation of the third rule. This means that we need to evaluate `From[Novel]@'C'`. The query is forwarded to peer C and the actual query execution starts from peer C. Relation `magic_Reach` collects the sources of the records in peer C based on backward recursive traversals. Then the contents of `magic_Reach` are used for forward traversals for the `Reach` predicate. □

In [22], we evaluated the performance of the magic set-based rewriting based on simple experiments. In summary, the result showed that the effectiveness of the rewriting depends on the target queries and the contents of the underlying databases. For example, the magic set-based rewriting of query Q5 shown above was not more efficient than the normal seminaive method-based execution because the backward traversals for `magic_Reach` are quite costly to find ancestors of the records in peer C. The result indicates that we need to construct a cost-based query optimization method that considers the properties of our framework. We leave the issue for future work.

6. Continual Query Processing

In this section, we describe the query processing strategy for continual queries.

6.1 Query Execution Strategy

As described in Sect. 4.2, a continual query is executed for a potentially long period of time, and is used for monitoring events occurred in the system. It is particularly useful in our context where information of record exchange is updated frequently in a distributed P2P network. The following example explains the idea of continual queries.

Example 8: Consider the following query Q6:

Query Q6

```
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 initial result `Query` = {(C)}. Assume that peer E copied the record from peer C. After that, the result will change as `Query` = {(E)}. □

As shown in this example, the result of a tracing query may change when an update is performed in the underlying databases. To monitor updates, we introduce the *continual execution mode*. We call a query executed in the continual execution mode a *continual query*. The query transformation steps are same as the ad-hoc execution mode. For query Q6, we derive seminaive method-based query fragments as shown in Example 4 for query Q1. The main differences are that `BReach`, `Origin`, and `From` are changed as `Reach`, `End`, and `To`, respectively. We omit the query fragments here.

Example 8 (Continued): We explain how query Q6 is executed as a continual query. First, the query is executed like

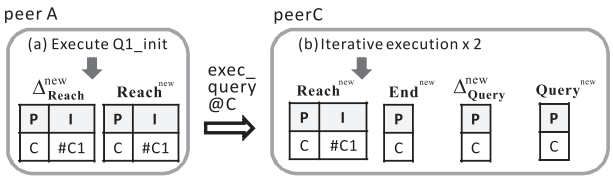


Fig. 14 Continual query execution: before update.

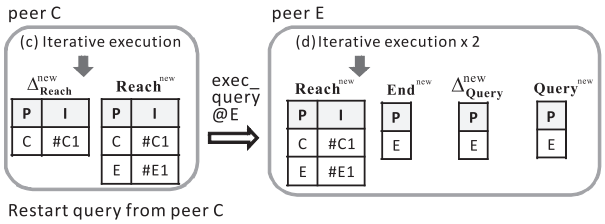


Fig. 15 Continual query execution: after update.

an ad-hoc query. Figure 14 shows the result for our example databases. The result `Query` = {(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 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, the following tuple insertions are performed in the local databases of peer C and E:

- `(#C1, E, #E1, ...)` in `To[Novel]@'C'`
- `(#E1, t1, a1)` in `Data[Novel]@'E'`
- `(#E1, C, #C1, ...)` in `From[Novel]@'E'`

The insertion of a new tuple in `To[Novel]@'C'` triggers a new query process. Our basic strategy is to restart the partial query from peer C by reevaluating Algorithm 1 in peer C. For that purpose, peer C needs to preserve $\Delta^{\text{new}}_{\text{Reach}}$ and $\texttt{Reach}^{\text{new}}$ obtained from peer A when the query was forwarded to peer C. By reevaluating the query from peer C, we get new results as shown in Fig. 15. Based on the reevaluation, we obtain a new result `Query` = {(E)}. □

As shown in this example, a continual query can be executed based on the seminaive method as in the case of ad-hoc queries. In summary, the query processing strategy is given as follows:

- If an update occurs in the local database of the initial peer, it executes the query as a new continual query.
- A non-initial peer processes the query as follows:
  - It stores the arguments (the query fragments $\{Q_S\}$ and the partial results $\{\Delta^{\text{new}}_S\}$ and $\{S^{\text{new}}\}$) passed from the previous peer persistently.
  - If an update occurs in its local database, the peer restarts the query using the stored partial results.

The idea is simple: we rederive the partial result from the peer in which an update occurs.

6.2 Relationship with Materialized View Maintenance

The continual query processing method described above is highly related with the *materialized view maintenance* problem [14] in deductive databases. For maintaining general recursive views, [15] proposed 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.

Fortunately, we can utilize a feature of our framework. In our framework, every update is handled as a tuple insertion. Depending on the update types, a record is inserted in each of the following local relations:

- record update in peer X: `Data@X` and `Change@X`
- record modification in peer X: `Data@X` and `Change@X`
- record deletion in peer X: `Change@X`
- record copy from peer X to peer Y: `To@X`, `From@Y`, and `Data@Y`

It means that the databases in our framework is *insertion-only*. Materialized view maintenance for insertions is fairly easy—it is known that we only have to run the seminaive method until the fixpoint [15]. The strategy shown above is based on this observation.

Finally we mention an improvement method for a restricted case. In the strategy shown above, we assume that a tracing query is written in datalog$^{\neg}$, which may contain negations. It requires a restart from an intermediate peer because a datalog query with negation is *non-monotonic*. In contrast, if a query does not include negations (e.g., query Q2), we can apply more efficient processing. The peer in which the update occurs only has to do an incremental query process using the current intermediate result instead of restarting. Since a query without negations is monotonic, the correctness of the result is assured. On the other hand, we cannot apply this incremental strategy to a query with negations. For example, if we apply the strategy to the example shown above, we get the wrong result `Query` = {(C, #C1), (E, #E1)}.

7. Discussion

7.1 Features of Our Research

In this paper, we described the query processing strategies for our traceable P2P record exchange framework. Here we summarize the features and contributions of this paper. The paper focuses on the query processing issues for our traceable P2P record exchange framework [20], [23], which is a unique approach to information exchange in a P2P network that incorporates the notion of data provenance. 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 [17].

The use of the datalog query language [2] is an another feature of the framework. A tracing process basically requires a recursive traversal along the path of record exchange. Datalog has an enough query representation power to describe such recursive processing requirements. We can write various types of tracing queries in a compact manner using datalog. In addition, we already have theoretical results on the query expressive power and query processing methods for deductive databases. In this paper, we set several constraints on allowable tracing queries. It enables clear and effective query transformation and executions while we still have enough query processing power for tracing.

7.2 Comparison with Declarative Networking

The most related work to our research is *declarative networking* [9], [26], [28]. In their research, a datalog-based recursive query processing framework used for collecting information from P2P networks and sensor networks. They propose query processing methods based on the seminaive method and the magic set method; the results greatly inspired our development. 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 three layer model introduced in Sect. 3.2. Datalog queries are used not only for describing high-level tracing requirements in the global layer, but also for representing distributed query execution in the local layer. 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. The framework of declarative network assumes that each peer has a special relation called `link`, which stores the information of neighborhood peers (or sensors). Given a monitoring query, the system constructs a dataflow graph by traversing `link` relations in distributed peers. When a new data item (e.g., a sensor measurement value) is detected, the data is propagated in the dataflow graph and finally collected in the initial sink. It means that the dataflow graph in declarative network is static, once an initial setup is finished. In our framework, on the other hand, `From` and `To` relations in each peer describe the record exchange information and the tracing process requires a traversal using the information stored in the relations. As we described in Sect. 6, the graph topology of the query process is dynamic because a new record copy changes the graph structure.
4. We proposed to use datalog$^\neg$ for representing tracing requirements. By allowing the use of negations, we can greatly improve the query expression power. We carefully restricted allowable tracing queries and examined the insertion-only feature of the underlying databases, and enabled a clear query processing framework. In contrast to our approach, declarative networking can only use non-negated datalog queries.

8. Conclusions and Future Work

In this paper, we discussed the details of query processing strategies for our P2P record exchange framework. The query language for writing a tracing query is datalog, a common database language with a recursive query processing facility. The datalog-based query specification allows us to write tracing queries in a compact manner, and a tracing query is evaluated by cooperating peers using query forwarding. The recursive nature of tracing is well suited to the deductive approach.

We presented two different query execution modes, the ad-hoc execution mode and the continual execution mode. The related queries are called ad-hoc queries and continual queries, respectively. We clearly defined valid tracing queries and some example queries were presented. We showed two different modes can be supported by a similar query processing framework based on the seminaive method, which is the most common query evaluation framework for datalog.

Our query processing framework is still in the development stage. We have the following future work:

- Cost-based query optimization: As described in Sect. 5.4.2, we may be able to present different query processing plans depending on queries. For selecting an optimal plan, we need to develop a cost model for query processing.
- Effective use of materialized view technologies: For tracing queries, especially for the queries asking past histories, materialized views [14] are quite helpful to reduce query response time. For that purpose, we need to develop a query processing method which effectively uses materialized views and a view selection and maintenance method which considers the trade-off of cost and benefit.
- Data replication and caching: Data replication and caching is helpful for efficient query processing and fault-tolerance. We already proposed some initial ideas in [21]. We would like to consider the problem in detail.
- Attachment of lineage information: The current framework takes a minimal approach. We do not maintain redundant information in the underlying databases. We may be able to improve the cost of query processing by attaching lineage information to each record. It will increase the storage cost, but some types of queries can be supported efficiently.
- Enhancement of the query language: Considering practical requirements of tracing, we need to incorporate additional features and constructs to our language. For

example, we can consider aggregate queries (e.g., for each record of peer A, count the number of peers that copied the record) and the temporal query facility because historical information can be considered as a temporal database. In addition, we may be able to introduce a stream database-like feature such as window-based queries into our continual queries.

- Support of different types of records: To simplify the problem, we only considered one type of records are shared among peers in the user layer. We may be able to extend to manipulate multiple types of records (e.g., `Novel` and `Author`). It means that the user layer behaves like a relational database. We may need to totally revise our query processing framework.
- Development of system implementation techniques: For an efficient implementation, we need to develop practical implementation techniques such as effective use of the query processing power of the underlying RDBMS in each peer.
- Experimental evaluation: We are currently developing a prototype system of our P2P record exchange framework. We would like to evaluate the effectiveness of our framework in terms of query response time, storage cost, and maintenance cost based on experiments. In [22], we already compared two popular query processing methods, the seminaive method and the magic set method, in our framework based on a simulation environment. The simulation environment should be extended considering real P2P networks, and we need to perform large-scale experiments.

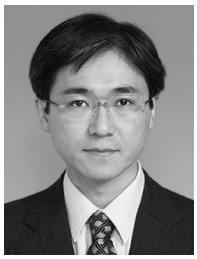
Acknowledgments

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

References

[1] K. Aberer and P. Cudre-Mauroux, "Semantic overlay networks," 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, vol.36, no.4, pp.335–371, 2004.
[4] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom, "ULDBs: Databases with uncertainty and lineage," Proc. VLDB, pp.953–964, 2006.
[5] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "An annotation management system for relational databases," Proc. VLDB, pp.900–911, 2004.
[6] BRICKS: Building resources for integrated cultural knowledge services, http://www.brickscommunity.org/
[7] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren, "Curated databases," Proc. ACM PODS, pp.1–12, 2008.
[8] P. Buneman and W.-C. Tan, "Provenance in databases (tutorial)," Proc. ACM SIGMOD, pp.1171–1173, 2007.
[9] T. Condie, D. Chu, J.M. Hellerstein, and P. Maniatis, "Evita raced: Metacompilation for declarative networks," VLDB, pp.1153–1165, 2008.
[10] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," Proc. VLDB, pp.471–480, 2001.
[11] Y. Cui, J. Widom, and J.L. Wiener, "Tracing the lineage of view data in a warehousing environment," ACM TODS, vol.25, no.2, pp.179–227, 2000.
[12] Gnutella, http://en.wikipedia.org/wiki/Gnutella
[13] T.J. Green, G. Karvounarakis, N.E. Taylor, O. Biton, Z.G. Ives, and V. Tannen, "Orchestra: Facilitating collaborative data sharing," Proc. ACM SIGMOD, pp.1131–1133, 2007.
[14] A. Gupta and I.S. Mumick eds, Materialized Views, MIT Press, 1999.
[15] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining views incrementally," Proc. ACM SIGMOD, pp.157–166, 1993.
[16] H. Gupta, X. Zhu, and X. Xu, "Deductive framework for programming sensor networks," Proc. ICDE, pp.281–292, 2009.
[17] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," Proc. ACM PODS, pp.1–9, 2006.
[18] Z. Ives, T.J. Green, G. Karvounarakis, N.E. Taylor, V. Tannen, P.P. Talukdar, M. Jacob, and F. Pereira, "The Orchestra collaborative data sharing system," ACM SIGMOD Record, vol.37, no.3, pp.26–32, Sept. 2008.
[19] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir, "Orchestra: Rapid, collaborative sharing of dynamic data," Proc. Conf. on Innovative Data Systems Research (CIDR 2005), pp.107–118, 2005.
[20] 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.
[21] F. Li, T. Iida, and Y. Ishikawa, "On physical organization of a P2P record exchange system," Proc. Forum on Data Engineering and Information Management (DEIM Forum 2009) (in Japanese), 2009.
[22] F. Li, T. Iida, and Y. Ishikawa, "'Pay-as-you-go' processing for tracing queries in a P2P record exchange system," Proc. DASFAA, vol.5463 of LNCS, pp.323–327, 2009. A long version is available from http://www.db.itc.nagoya-u.ac.jp/papers/2009-dasfaa-li-long.pdf
[23] F. Li and Y. Ishikawa, "Traceable P2P record exchange based on database technologies," Proc. APWeb, vol.4976 of LNCS, pp.475–486, 2008.
[24] L. Liu, C. Pu, and W. Tang, "Continual queries for internet scale event-driven information delivery," IEEE TKDE, vol.11, no.4, pp.610–628, 1999.
[25] M. Liu, N.E. Taylor, W. Zhou, Z.G. Ives, and B.T. Loo, "Recursive computation of regions and connectivity in networks," Proc. ICDE, pp.1108–1119, 2009.
[26] 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," Proc. ACM SIGMOD, pp.97–108, 2006.
[27] "Orchestra: Managing the collaborative sharing of evolving data," http://www.csi.upenn.edu/~zives/orchestra/
[28] P2: Declarative networking, http://p2.berkeley.intel-research.net/
[29] W.-C. Tan, "Research problems in data provenance," IEEE Data Engineering Bulletin, vol.27, no.4, pp.45–52, 2004.
[30] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," Proc. Conf. on Innovative Data Systems Research (CIDR 2005), pp.262–276, 2005.
[31] C. Zaniolo, "The logic of query languages," in Advanced Database Systems, ed. C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari, chapter 8, pp.163–199, Morgan Kaufmann, 1997.
[32] W. Zhou, Y. Mao, B.T. Loo, and M. Abadi, "Unified declarative platform for secure networked information systems," Proc. ICDE, pp.150–161, 2009.

Fengrong Li is a Ph.D. candidate majoring in Systems and Social Informatics at Graduate School of Information Science in Nagoya University. Her main research interests lie in data provenance, P2P database, and integration of distributed heterogeneous information. She is a student member of DBSJ and IPSJ.



Yoshiharu Ishikawa is a Professor at Information Technology Center, Nagoya University. 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, IPSJ, JSAI, ACM, and IEEE.