An Enhanced RETE Algorithm for Large Scale Data Access

Nobuhiro Ajitomi, Hiroyasu Kurose

Software Research Center
RICOH Co., Ltd.
1-1-17, Koishikawa, Bunkyo-ku, Tokyo, 112 JAPAN

Abstract

This paper introduces the OD-RETE pattern matching algorithm which is obtained by incorporating the on-demand evaluation mechanism into the RETE algorithm and which is suitable for production systems accessing to large scale data. It allows the construction of both a "data-driven" network simulating an ordinary RETE network and a "request-driven" network performing on-demand formula evaluation. The latter network can treat large data without loading the entire data into the network. This paper describes the OD-RETE algorithm by defining nodes as "objects" which exchange messages between one another, it also discusses the ways to utilize this algorithm with DBMSs.

Keywords: RETE algorithm, object-oriented programming

1 Introduction

RETE[8] is a well known pattern matching algorithm. It is efficient for comparing a large collection of patterns with a large collection of data. Many production systems such as OPS5[5,8] use this algorithm or one of its variants[9,15,19]. It is used to determine which rules satisfy condition parts.

Sharing pattern matching tests and storing the results of the tests make the RETE algorithm efficient. The RETE algorithm which has these features is useful not only for production systems but also for database management systems. It can be used to validate integrity constraints of a database management system (DBMS) and to evaluate queries.

The RETE algorithm has been seldom used with a DBMS. With a DBMS, the original RETE algorithm has several problems such as memory shortage and evaluation time explosion. The RETE algorithm requires a very large memory area for the working memory (WM) because the records in a DBMS must be loaded into the WM. There are some attempts[10] in which only activated data are loaded in order to save the WM area.

The network in the RETE algorithm is reconstructed when rules are changed. It is difficult to keep the network and the WM consistent after reconstruction. Several production systems support incremental compilation of a RETE network [3,5,15]. It is not a crucial problem that the WM is cleared in reconstruction as far as a RETE network is seldom reconstructed. However it may be a severe problem when the network is often reconstructed or the recomputation cost of the WM is intolerable. The RETE algorithm running with a DBMS has a large WM, therefore re-evaluation will require a great deal of computation.

If the above problems are solved, the RETE algorithm increases efficiency of database queries and expert systems that use very large scale data. It is also applied to view update[4,12,17], snapshot update[1,11] and multiple query optimization[6,18]. It is also useful for inference procedures on a database such as checking database integrity [7,14].

The RETE algorithm works effectively only when data in a RETE network change relatively slowly. Because the RETE algorithm maintains states between production cycles, it is inefficient in situations where most of the data change in every cycle. Intermediate results of matching can be stored in a RETE network so that all data in the WM need not be recomputed every time the WM is modified. But if there are a lot of data, the size of intermediate results may be combinatorially explosive. An algorithm named TREAT in which pattern matching is recomputed every cycle without intermediate results is proposed[13]. TREAT is better than RETE when data change frequently. An algorithm that stores or recomputes the intermediate results needed is better than RETE and TREAT.

We have solved these problems by adding special functions to each node in a RETE network. Our enhanced al-
algorithm is called OD-RETE algorithm, where ‘OD’ stands for On-Demand evaluation mechanism.

In the OD-RETE algorithm, nodes in a network are regarded as ‘objects’ (in the sense of object-oriented programming paradigm), or processes communicating to one another. A node can send a message to another node. A node that receives a message will take some actions and send a reply to the message sender. All interactions between nodes can be handled as messages. A downward data flow (seen in RETE) is simulated by “feed data” messages that are sent from upward nodes to downward nodes. A node can request to retransmit all data by sending a “request all data” message to its upward node. In general, data feeding messages flow downwards and data requesting messages flow upwards. An OD-RETE network can behave as an ordinary RETE network with using the former feature, and also perform on-demand formula evaluation, as a DBMS does, with using the latter feature. These features improve the efficiency of memory use and increase the computing speed.

The structure of this paper is as follows. In the next section, the OD-RETE algorithm is defined. In section 3, the features of this algorithm are described. In section 4, applications for a DBMS are discussed. Section 5 concludes this paper and presents a discussion of future works.

2 Algorithm Description

This section describes the OD-RETE algorithm. An OD-RETE network is constructed from three types of nodes which are test nodes, join nodes, and memory nodes. Test and join nodes correspond to one-input nodes and two-input nodes in the original RETE algorithm [8] respectively. Memory nodes correspond to left- and right-memories of two-input nodes in [8]. A memory node has data storages, called token memories, each of which can contain a token sequence. This section describes how these three nodes in our algorithm work.

Like an ordinary RETE network, an OD-RETE network has several entries (from which the network receives tokens from the outer world) and exits (i.e., goals, each of which corresponds to each production rule). In this paper, downward direction means the direction from entries to exits, and upward direction means the opposite direction.

2.1 Notation

This section introduces some concepts and notations that are necessary to describe our algorithm.

Let U be the set of all values we are concerned with. When a set of attributes A is given, a tuple on A is defined as a mapping from A to U. We represent the set of all tuples on A by U^A. When A is empty, there exists only one tuple, denoted by 1.

A token is a signed tuple as defined in [8]. Tokens +x and -x denote an addition and a deletion of x respectively. A sequence of signed tuples on A is called a token sequence on A. In this paper, a token sequence X = (±x1, ±x2, . . .) is expressed by a summation form such as X = ±x1 ± x2 ± . . . = ∑caxa (where ca = ±1). When a token sequence X can be expressed as X = x1 + x2 + . . . = ∑xα (where xα ≠ xβ for all different α and β), X is interpreted as a relation X = {x1, x2, . . .}. An empty sequence, denoted by 0, is regarded as an empty relation.

The set of all values we are concerned with is denoted by U. We represent the set of all tuples on A by UA. When A is empty, there exists only an empty sequence, denoted by 0. A memory node corresponds to left- and right-memories of two-input nodes in [8]. A memory node has data storages, which correspond to each production rule. In this paper, a memory node is called a forgiven memory, each of which can contain a token sequence, called a token, which corresponds to each production rule. In this paper, a token sequence X = Ccα xα is called a token sequence.

This section gives descriptions of messages that are exchanged between nodes in the OD-RETE algorithm. Those messages can be classified into three groups: network manipulating messages, data feeding messages, and data requesting messages. Network manipulating messages (Minit, Mfree, Mconnect, and Mdisconnect) are used to grow or shrink networks. Data feeding messages (Mfeed) are used to send data downwards, as in a RETE network. Data requesting messages (Mrequest and Mresponse) are used to request data. Data requesting messages flow upward and replies flow downward in a network.

Minit(R) ——— Where R is a node. This message requests the receiver node to connect its first upward port to the
appropriate downward port of \( R \). This is used to add a test or memory node to a network. (A test/memory node has one upward port.)

\[ M_{\text{node}}(R_1, R_2) \] — Where \( R_1 \) and \( R_2 \) are nodes. This message requests the receiver node to connect its first and second upward ports to the appropriate downward ports of \( R_1 \) and \( R_2 \). This is used to add a join node to a network. (A join node has two upward ports.)

\[ M_{\text{connect}}(R, m) \] — Where \( R \) is a node and \( m \) is an upward port number of \( R \). This message requests to connect \( (m|R) \) to some downward port in the message receiver. The receiver returns the downward port number.

\[ M_{\text{discon}}(i) \] — Where \( i \) is a downward port number of the message receiver. This message requests to disconnect a connection on a downward port \( i \).

\[ M_{\text{feed}}(m, X) \] — Where \( m \) is an upward port number of the message receiver and \( X \) is a token sequence. This message is used to send \( X \) to the receiver node through its upward port \( m \).

\[ M_{\text{reqold}}(i, X) \] — Where \( i \) is a downward port number of the message receiver node and \( X \) is a relation. If the downward port \( i \) is defined to send tokens on \( A \), \( X \) must be a relation on a subset of \( A \). When \( X = 1 \), this message requests to transmit all data that have been sent through the downward port \( i \) of the receiver node. Sending this message simulates a memory-scan operation. When \( X \neq 1 \), \( XY \) will be returned, where \( Y \) is the returned value for \( M_{\text{reqold}}(i, 1) \). Note that when \( M_{\text{reqold}} \) is sent twice the second execution will return the same value as the first one (if there are no other activities in the network).

\[ M_{\text{reqnew}}(i) \] — Here \( i \) is a downward port number of the message receiver node. This message requests to transmit all data that have arrived at the receiver node and that have not been sent through the downward port \( i \). Those data are sent through the downward port \( i \). Note that when \( M_{\text{reqnew}}(i) \) is sent twice the second execution will return 0, i.e., an empty token sequence (if there are no other activities in the network).

2.3 Nodes

We give complete definition of OD-RETEnodes here. First we explain variables allocated and maintained by each node. A node has several variables in it which contain the node's state information. These variables are listed below.

Upward/downward ports — An upward port is implemented as a pair of two variables \( R^p \) and \( m^p \) (where \( i = 1 \) in a test/memory node; \( i = 1, 2 \) in a join node). Similarly, a downward port is implemented as

\[ \text{Feed}(i, X) = \{ (R^p, M_{\text{reqnew}}(X)) (X \neq 0) \} \]

\[ \text{ReqOld}(i, X) = \{ [R^p, M_{\text{reqold}}(m^p, X)] \} \]

\[ \text{ReqNew}(i) = \{ [R^p, M_{\text{reqnew}}(m^p)] \} \]

Figure 1: Procedures used in node descriptions.

A pair of \( R^p \) and \( m^p \) (where \( i = 1, 2, \ldots \)). When there exists a connection between \( R|m \) and \( (n|S) \), \( R^p = R \) and \( m^p = m \) hold in \( S \).

Token memories — A memory node has one upward token memory \( X^p \) and several downward token memories \( X^p_i, X^p_i, \ldots \). For every memory corresponds a port. (A memory node has one upward port and several downward ports.) The \( X^p \) memory keeps the sum of all tokens that have arrived at the memory node and the \( X^p_i \) memory keeps the sum of all tokens that have departed from the \( i \)-th downward port. Since these token memories can be implemented to share stored data, the amount of the stored data is proportional to \( \| \text{Abs}(X^p) \| \text{Abs}(X^p_1) \| \text{Abs}(X^p_2) \| \ldots \| \) which is less than \( \| \text{Abs}(X^p) \| + \| \text{Abs}(X^p_1) \| + \| \text{Abs}(X^p_2) \| \ldots \| \).

Working variables — Some working areas are temporarily needed.

The self node — In the description of each node, the node itself is referred as a special variable \( R^p \).

For ease of node description, we define three procedures as shown in Fig. 1. Procedure \( \text{Feed}(i, X) \) feeds data \( X \) to \( R^p \); procedure \( \text{ReqOld}(i, X) \) requests \( (i|R^p) \) to retransmit all data: procedure \( \text{ReqNew}(i) \) requests \( (i|R^p) \) to transmit new data.

Test node — Let \( A \) be a set of attributes, and \( \psi \) be a logical formula. The test node for \( \psi \) restricts a token sequence \( A \) by \( \psi \). The description of this test node is shown in Fig. 2. When receiving \( M_{\text{reqold}}(1, X) \), this node behaves like a one-input node in [8], i.e., this node feeds \( \text{Restrict}(\psi, X) \) to all downward nodes. When receiving \( M_{\text{reqold}} \) or \( M_{\text{reqnew}} \), this node forwards this message to its upward node, restricts the value returned from the upward node by \( \psi \), and replies the restricted data.

Join node — Let \( A_1, A_2 \) be sets of attributes. The join node on \( A_1 \) and \( A_2 \) joins two token sequences on \( A_1 \) and \( A_2 \) and generates a new token sequence on \( A_1 \cup A_2 \). For \( M_{\text{feed}}(i, X) \), this node behaves like a two-input node in [8], except that it sends \( M_{\text{reqold}} \) message upward instead of scanning left- or right-memory. For \( M_{\text{reqold}}(i, X) \) and
3 Features and Usage

This section shows how networks are constructed from nodes defined in the previous section.

3.1 Data-Driven Network

Let us suppose there are relations $X_1^{th}, X_2^{th}, \ldots$ on $A_1, A_2, \ldots$, and $A \subset U_A$. When a logical formula $\psi$ is given, a query by $\psi$ means to compute $X_+ = \{z \in U_A | z$ satisfies $\psi; z|_A \in X_1^{th}$ holds for all $i\}$ from $X_1^{th}, X_2^{th}, \ldots$. If the formula $\psi$ is simple enough, it is possible to construct a RETE network that processes this query. This RETE network can be considered as a data-driven computing system. Since each node is programmed to take an action when a token arrives at the node, the whole computation process will be completed just after tokens are fed to the network.

The OD-RETE algorithm is designed as an extension of the RETE algorithm, and it is possible to construct a data-driven OD-RETE network as follows. First, a network is constructed from the given condition $\psi$ according to the ordinary RETE algorithm; test nodes are used instead of one-input nodes, join nodes connected to non-buffering memory nodes are used, i.e.: $\psi$. The next step is to add a buffering memory node $R_x^{th}$ to the exit node by sending $M_{\text{read}}(R_x^{th})$ message to $R_{xt}^{th}$. The final step is to execute:

\[ m_x^{th} \leftarrow [R_{xt}^{th}, M_{\text{connect}}(R_{xt}^{th}, 1)] \]

Instead of two-input nodes (where $R_1^{th}$ and $R_2^{th}$ are non-buffering memory nodes, and $R_1^{th}$ is a join node). The network will have several ports of entry nodes $(m_1^{th}, m_2^{th}, \ldots)$ that correspond to $X_1^{th}, X_2^{th}, \ldots$ respectively, and one exit node $R_1^{th}$ that corresponds to $X_+$. The next step is to add a buffering memory node $R_x^{th}$ to the exit node by sending $M_{\text{read}}(R_x^{th})$ message to $R_{xt}^{th}$. The final step is to execute:

\[ m_x^{th} \leftarrow [R_{xt}^{th}, M_{\text{connect}}(R_{xt}^{th}, 1)] \]

instead of two-input nodes (where $R_1^{th}$ and $R_2^{th}$ are non-buffering memory nodes, and $R_1^{th}$ is a join node). The network will have several ports of entry nodes $(m_1^{th}, m_2^{th}, \ldots)$ that correspond to $X_1^{th}, X_2^{th}, \ldots$ respectively, and one exit node $R_1^{th}$ that corresponds to $X_+$. The next step is to add a buffering memory node $R_x^{th}$ to the exit node by sending $M_{\text{read}}(R_x^{th})$ message to $R_{xt}^{th}$. The final step is to execute:

\[ m_x^{th} \leftarrow [R_{xt}^{th}, M_{\text{connect}}(R_{xt}^{th}, 1)] \]

instead of two-input nodes (where $R_1^{th}$ and $R_2^{th}$ are non-buffering memory nodes, and $R_1^{th}$ is a join node). The network will have several ports of entry nodes $(m_1^{th}, m_2^{th}, \ldots)$ that correspond to $X_1^{th}, X_2^{th}, \ldots$ respectively, and one exit node $R_1^{th}$ that corresponds to $X_+$. The next step is to add a buffering memory node $R_x^{th}$ to the exit node by sending $M_{\text{read}}(R_x^{th})$ message to $R_{xt}^{th}$. The final step is to execute:

\[ m_x^{th} \leftarrow [R_{xt}^{th}, M_{\text{connect}}(R_{xt}^{th}, 1)] \]
(X^{db}_1) \rightarrow (1)R^{init}_{x^{db}_1}[1] \rightarrow (1)R^{mem}_{x^{db}_1}[1] \rightarrow (1) R^{init}_{P_1(A_1) \wedge P_2(A_2) \wedge Q(A)}.

Figure 7: A data-driven network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$.

\[
\begin{align*}
(X^{db}_2) & \rightarrow (1)R^{init}_{x^{db}_2}[1] \rightarrow (1)R^{mem}_{x^{db}_2}[1] \rightarrow (1) R^{init}_{P_1(A_1) \wedge P_2(A_2) \wedge Q(A).}
\end{align*}
\]

Figure 8: A data-driven network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ and $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$.

$R^{exit}_{x^{db}}$ for $X_\psi$ is obtained. The usage of this network is shown in Fig.9.

In this network, requesting data to $R^{exit}$ triggers all necessary computations and causes $X^{db}_{i+1}$ to be updated. This happens in the initialization step and the updating step by $R^{exit}$. If $X^{db}_i$ in each $R^{init}$ is implemented as a reference to a relation $X^{db}_{i+1}$ in a DBMS, the initialization step is simplified as Fig.10, which is the same as the initialization step of a data-driven OD-RETE network as shown in Fig.6.

Fig.11 is a request-driven OD-RETE network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ and $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$.

Networks discussed in the previous section and here are to be called totally-data-driven OD-RETE networks and totally-request-driven OD-RETE networks. There are various way to construct networks besides these two extreme strategies, since it is possible to make partially-data-driven/partially-request-driven OD-RETE networks in which both of data-driven subnetworks and request-driven subnetworks exist. Such mixed OD-RETE networks can be used to integrate production systems and DBMSs.

Fig.12 is an example of the mixed network. Here $R^{exit}$ is a non-buffering memory node, and $R^{init}$ and $R^{mem}$ are buffering memory nodes. Suppose $X^{db}_i$ is a large relation in a DBMS. When a token $t^{i+1}$ is fed to $R^{init}$, $R^{init}$ receives $M_{request}(1,y)$ (where $y = x_{i+1}|_{A_1 \cup A_2}$) and will return $yX^{down}$. If $X^{down}$ has an index whose key is $A_1 \cap A_2$, $yX^{down}$ can be retrieved easily by utilizing the index. Thus, only required data in $X^{db}_i$ flow into the network.

3.3 Dynamic Network Construction

In section 3.1 and 3.2, networks are constructed statically, i.e., no tokens are fed to a network during network construction. However, our OD-RETE algorithm allows to feed tokens to a network under the construction. Networks

\[
\begin{align*}
(X^{db}_1) & \rightarrow (1)R^{init}_{x^{db}_1}[1] \rightarrow (1)R^{mem}_{x^{db}_1}[1] \rightarrow (1) R^{init}_{P_1(A_1) \wedge P_2(A_2) \wedge Q(A)}.
\end{align*}
\]
can grow (or shrink) even after they have been initialized and have been fed data.

For any node $R$ in an OD RETE network, it is possible to grow the network by sending $M_{test}(R)$ message to a new test/memory node $S$. Similarly, for any nodes $R_1, R_2$ in an OD-RETE network, it is possible to grow the network by sending $M_{test}(R_1, R_2)$ message to a new join node $S$. The newly created part of the network is initialized by $M_{reqo1}$ message toward the downward end, same as in Fig.6 and Fig.10. Once initialized, there are no differences between the older part and the newly created part of the network. Thus, in our algorithm, usage of a network is independent of the way how the network has been constructed, in data-driven manner or request-driven manner, statically or dynamically, etc.

For example, the network in Fig.7 can grow up to the network in Fig.8 by $[R_{test}^1, M_{test}(1, 1)]$ and $[R_{test}^2, M_{test}(R_{test}^2)]$. The newly created part is initialized by $X_{test} \leftarrow R_{test}^1 \cdot M_{test}(1, 1)]$. A network can shrink by an $M_{test}$ message. A node $R$ is removable when $R$ has no downward nodes (i.e., $R_{test}^i = 0$ for all $i$). The $M_{test}$ message to a removable node removes the node from a network. If $S$ is the only downward node of $R$, removing $S$ makes $R$ removable. Fig.13 is a definition of the “recursive free” message to remove the receiver node and all other nodes that become removable.

4 Application for DBMS

OD-RETE may be applied to several kinds of problems for which RETE is inefficient. The OD-RETE algorithm loads data into the WM as little as possible. It can be applied to rule base with very large data. Especially, it is useful for productions that refer a lot of objects in a database.

In RETE, even invariable records such as code tables, must be loaded into the WM. OD-RETE makes the WM small. It also improves the speed of inferences. Therefore, inferences on a database can be executed with OD-RETE at an enough speed. Checking database integrity is an example of productions on a database.

OD-RETE can connect and disconnect its network without clearing the WM. It can add or delete rules dynamically. The property of the dynamic reconstruction is useful for query optimization of a database.

Checking integrity, materialized view, and query optimization are illustrated in this order as examples of its applications.

4.1 Checking Database Integrity

Database integrity is concerned with ensuring that the database is correct even though users or application programs try to incorrectly modify it. It is desirable to be able to describe integrity constraints on any combination of records. Most DBMSs, however, allow users to constrain only values of fields and referential integrities.

In order to check the database integrity, an OD-RETE network is constructed from the negation of the integrity constraint. The network has a buffered memory node as its exit node. While the database satisfies the integrity constraint, no tokens reach the exit node of the network. At the end of each transaction, the exit node is checked whether its memory is empty or not. If the memory is not empty, the integrity constraint is not satisfied in the current state and its corresponding transaction must be rollbacked.

$$M_{test}$$ : if there exists $i$ such that $R_{test}^i \neq 0$
then return immediately
$$(R_{test}^i, M_{test}(i, 1)), (R_{test}^i, M_{test})$$
(for all upward port $i$)

Figure 13: $M_{test}$ message to recursively free nodes.
4.2 Materialized View

View materialization refers to the process of evaluating a view definition and storing its results. The resulting data is called a materialized view. As the database changes because of updates applied to the base relations, the materialized view also changes. A materialized view can always be brought up to date by re-evaluation. However, complete re-evaluation is often inefficient, and the cost involved may be intolerable. The problem addresses how to quickly update a view in response to database changes. This problem is different from the traditional view update problem. Users can only update base relations, and direct update to views is not considered.

It is possible to construct an OD-RETE network from the condition that defines a view. If this network has a memory node as its exit node, it works as a materialized view. Access to the materialized view is performed by sending a message to the memory node.

For example, Fig. 14 shows a network for multiple views. In this figure, $X^b_h$ is a database relation. $R^h$ is a non-buffering memory node. $R^{mem}$ is a buffering memory node. The conditions corresponding to the views are as follows:

$X^b_1 \rightarrow X^{view}_1, X^{view}_2 \rightarrow X^{view}_1, X^{view}_3 \rightarrow X^{view}_1$

$X^b_2 \rightarrow X^{view}_2, X^{view}_3 \rightarrow X^{view}_2, X^{view}_4 \rightarrow X^{view}_2$

In this network, requesting data to $R^{mem}$ only causes the access to the memories in $R^{mem}$ because $R^{mem}$ is recomputed when a tuple is added to or deleted from $X^b_h$. $X^{view}$ is accessed as fast as an ordinary materialized view.

When a database is updated, new tuples are added to or deleted from $X^b_h$. $X^{view}$ is updated differentially. This is faster than complete re-evaluation. Multiple views can share intermediate results of subconditions. The total throughput of multiple view evaluation is higher than that of individual evaluation of views.

When a view is added or deleted, the OD-RETE network is reconstructed. The network is reconstructed dynamically without interfering another view access. Even after the reconstruction, the speed of view access does not slow down. A partial change of view schemas does not cause complete re-evaluation of the whole view.

4.3 Query Optimization

The method of view optimization applies to ordinary query optimization. The result of queries is stored in an OD-RETE network in the same way as the materialized views. The result gives a possibility of improvement of response time of DBMSs. However, there are a lot of various queries in retrievals. It is impossible to build the network in advance, because we cannot predict all the queries. In each retrieval, the subnetwork corresponding to the query is added to the OD-RETE network. After a retrieval is executed, identical retrievals are executed only by accessing to the WM. It is useful for queries which are partially equivalent to previous ones.

The method is also applicable to multiple query optimization. If there are queries that contain common subqueries, it is not efficient to process them individually. The process can share intermediate results of the subpatterns.

By the existence of various patterns of retrievals, the network may grow unlimitedly. This explosion can be avoided by removing rarely used subnetworks using strategies such as LRU. The removal causes no effect to other parts of the network. Queries that are not related to the removed subnetworks can be answered quickly after the removal.

5 Concluding Remarks

A new approach, the OD-RETE algorithm, has been proposed to adapt the RETE algorithm to DBMSs. This algorithm has been obtained by incorporating an on-demand evaluation mechanism into the RETE algorithm and treating all control/data flows as "messages" exchanged between RETE nodes.

The OD-RETE algorithm is an extension of the original RETE algorithm and has all advantages that the RETE algorithm has. Besides those advantages, the OD-RETE algorithm has the "request-driven" feature. This feature enables to avoid feeding all contents of a database into an OD-RETE network. It also enables incremental query addition/deletion. The OD-RETE algorithm, which has the "data-driven" feature that comes from the RETE algorithm and the "request-driven" feature that fits DBMSs, is a relevant solution for integrating production systems and DBMSs.
Finally we mention future plans relating to this research. The first plan is to implement an experimental OD-RETE-based production system to evaluate the OD-RETE algorithm. The second is to enhance the algorithm and to make it possible to remove and reconstruct a branch of a network while the branch is in use. The last is to incorporate an on-demand evaluation mechanism into other extensions of the RETE algorithm. Specifically we plan to integrate the OD-RETE and an enhanced RETE algorithm that is designed to treat quantified logical formulas [2].

6 Acknowledgements

The authors would like to express their gratitude to Dr. Hideko Kunii for her invaluable suggestions. The authors would also like to thank Mr. Katsumi Kanasaki. The authors are grateful to Ms. Naoko Ichikawa and Mr. Hideaki Nakayama for their helpful assistance and kind criticism.

References