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 untolerable. 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-

gorithm 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 $X = \langle \pm x_1, \pm x_2, \ldots \rangle$ is expressed by a summation form such as $X = \pm x_1 \pm x_2 \pm \ldots = \sum c_\alpha x_\alpha$ (where $c_\alpha = \pm 1$). When a token sequence X can be expressed as $X = x_1 + x_2 + \ldots = \sum x_\alpha$ (where $x_\alpha \neq x_\beta$ for all different α and β), X is interpreted as a relation $X = \{x_1, x_2, \ldots\}$. An empty sequence, denoted by 0, is regarded as an empty relation.

The *product* of two token sequences $X_1 = \sum c_{1\alpha} x_{1\alpha}$ and $X_2 = \sum c_{2\alpha} x_{2\alpha}$ is defined by $X_1 X_2 = \sum (c_{1\alpha} c_{2\beta}) y_{\alpha\beta}$ (where $y_{\alpha\beta} = x$ if $\{x_1\} \bowtie \{x_2\} = \{x\}$; $y_{\alpha\beta} = 0$ when $\{x_1\} \bowtie \{x_2\} = \{\}$). Obviously $1X = X1 = X$ and $0X = X0 = 0$ hold for any X. When all $c_{i\alpha}$ is 1, the product XY is equivalent to the natural join of two relations, i.e., $X \bowtie Y$.

For a token sequence $X = \sum c_\alpha x_\alpha$ and a logical formula ψ, X*'s restriction by* ψ is defined by $\mathsf{Restrict}_\psi(X) = \sum c'_\alpha x_\alpha$ (where $c'_\alpha = c_\alpha$ if x_α satisfies ψ; $c'_\alpha = 0$ otherwise).

For a token sequence $X = \sum c_\alpha x_\alpha$ (where $x_\alpha \neq x_\beta$ for all different α and β, and $c_\alpha \neq 0$ for all α), its *absolute value* is defined by $\mathsf{Abs}(X) = \sum x_\alpha$. Note that $\mathsf{Abs}(X)$ is a relation.

When x is a tuple on A and A' is a subset of A, its *projection* $x' = x|_{A'}$ is a tuple on A' that satisfies $x'(a) = x(a)$ for all $a \in A'$. For a relation $X = \sum x_\alpha$ on A, X's projection is defined by $X|_{A'} = \mathsf{Abs}(\sum (x_\alpha|_{A'}))$.

A *message expression* $[R, M]$ denotes an action to send a message M to a node R and to await its reply, and the returned value. A message expression (R, M) denotes an action to send a message M to a node R. In (R, M), the message sender does not wait for a reply, the returned value is discarded. For ease of explanation, we introduce a dummy null node, denoted by 0. $[0, M]$ returns 0 and $(0, M)$ means no operation for any message M.

Each node has several *upward ports* and *downward ports* which contain information about the network structure. We represent the i-th upward port and the j-th downward port of a node R by $\langle i|R$ and $R|j\rangle$ respectively. An upward port can be connected to a downward port of another node, and a downward port can be connected to an upward port of another node. We represent a connection $R|i\rangle$ between $\langle j|S$ by a diagram:

$$R|i\rangle \longrightarrow \langle j|S$$

2.2 Messages

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 (M_{init}, M_{free}, M_{connect}, and M_{discon}) are used to grow or shrink networks. Data feeding messages (M_{feed}) are used to send data downwards, as in a RETE network. Data requesting messages (M_{reqold} and M_{reqnew}) are used to request data. Data requesting messages flow upward and replies flow downward in a network.

$M_{\mathrm{init}}(R)$ —— 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.)

$M_{\text{init}}(R_1, R_2)$ —— 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_{free} —— When this message is sent to a node in a network the receiver node is removed from the network.

$M_{\text{connect}}(R, m)$ —— Where R is a node and m is an upward port number of R. This message requests to connect $\langle m|R$ 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_{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-RETE nodes 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_i^{up} and m_i^{up} (where $i = 1$ in a test/memory node; $i = 1, 2$ in a join node). Similarly, a downward port is implemented as a pair of R_i^{down} and m_i^{down} (where $i = 1, 2, \ldots$). When there exists a connection between $R|m\rangle$ and $\langle n|S$, $R_m^{\text{down}} = S$ and $m_m^{\text{down}} = n$ hold in R, and $R_n^{\text{up}} = R$ and $m_n^{\text{up}} = m$ hold in S.

$$
\begin{aligned}
\mathsf{Feed}(i, X) &= \begin{cases} (R_i^{\text{down}}, M_{\text{feed}}(X))(X \neq 0) \\ \text{do nothing} \qquad (X = 0) \end{cases} \\
\mathsf{ReqOld}(i, X) &= [R_i^{\text{up}}, M_{\text{reqold}}(m_i^{\text{up}}, X)] \\
\mathsf{ReqNew}(i) &= [R_i^{\text{up}}, M_{\text{reqnew}}(m_i^{\text{up}})]
\end{aligned}
$$

Figure 1: Procedures used in node descriptions.

Token memories —— A memory node has one upward token memory X_1^{up} and several downward token memories $X_1^{\text{down}}, X_2^{\text{down}}, \ldots$. For every memory corresponds a port. (A memory node has one upward port and several downward ports.) The X_1^{up} memory keeps the sum of all tokens that have arrived at the memory node and the X_i^{down} 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 $\|\mathsf{Abs}(X_1^{\text{up}}) \cup (\mathsf{Abs}(X_1^{\text{down}}) \cup \mathsf{Abs}(X_2^{\text{down}}) \cup \ldots)\|$ which is less than $\|\mathsf{Abs}(X_1^{\text{up}})\| + (\|\mathsf{Abs}(X_1^{\text{down}})\| + \|\mathsf{Abs}(X_2^{\text{down}}\| \ldots)$, where $\|X\|$ denotes the number of tuples contained in a relation X.

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^{self}.

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

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

$M_{\text{init}}(R)$: $R_1^{\text{up}} \leftarrow R,\ m_1^{\text{up}} \leftarrow [R, M_{\text{connect}}(R^{\text{self}}, 1)]$
M_{free} : $[R_1^{\text{up}}, M_{\text{discon}}(m_1^{\text{up}})]$
$M_{\text{connect}}(R, m)$: find i such that $R_i^{\text{down}} = 0$
$R_i^{\text{down}} \leftarrow R,\ m_i^{\text{down}} \leftarrow m$
reply i
$M_{\text{discon}}(i)$: $R_i^{\text{down}} \leftarrow 0$
$M_{\text{feed}}(1, X)$: $\mathsf{Feed}(i, \mathsf{Restrict}_\psi(X))$ (for all i)
$M_{\text{reqold}}(i, X)$: reply $\mathsf{Restrict}_\psi(\mathsf{ReqOld}(1, X))$
$M_{\text{reqnew}}(i)$: $X \leftarrow \mathsf{Restrict}_\psi(\mathsf{ReqNew}(1))$
$\mathsf{Feed}(j, X)$ (for all $j \neq i$)
reply X

Figure 2: Test node for ψ.

$M_{\text{init}}(R_1, R_2)$: $R_1^{\text{up}} \leftarrow R_1, m_1^{\text{up}} \leftarrow [R_1, M_{\text{connect}}(R^{\text{self}}, 1)]$
$R_2^{\text{up}} \leftarrow R_2, m_2^{\text{up}} \leftarrow [R_2, M_{\text{connect}}(R^{\text{self}}, 2)]$
M_{free} : $[R_1^{\text{up}}, M_{\text{discon}}(m_1^{\text{up}})]$
$[R_2^{\text{up}}, M_{\text{discon}}(m_2^{\text{up}})]$
$M_{\text{connect}}(R, m)$: same as M_{connect} for a test node
$M_{\text{discon}}(i)$: same as M_{discon} for a test node
$M_{\text{feed}}(i, X)$: $Y \leftarrow \mathsf{ReqOld}(2-i, \mathsf{Abs}(X)|_{A_1 \cap A_2})$
$\mathsf{Feed}(j, XY)$ (for all j)
$M_{\text{reqold}}(i, X)$: reply $X\,\mathsf{ReqOld}(1, X|_{A_1})\,\mathsf{ReqOld}(2, X|_{A_2})$
$M_{\text{reqnew}}(i)$: $X_1 \leftarrow \mathsf{ReqNew}(1)$
$Y_2 \leftarrow \mathsf{ReqOld}(2, \mathsf{Abs}(X_1)|_{A_1 \cap A_2})$
$X_2 \leftarrow \mathsf{ReqNew}(2)$
$Y_1 \leftarrow \mathsf{ReqOld}(1, \mathsf{Abs}(X_2)|_{A_1 \cap A_2})$
$\mathsf{Feed}(j, X_1Y_2 + X_2Y_1)$ (for all $j \neq i$)
reply $X_1Y_2 + X_2Y_1$

Figure 3: Join node for A_1 and A_2.

$M_{\text{reqnew}}(i)$ messages, the node generates data requesting messages to compute $X\,\mathsf{ReqOld}(1,1)\,\mathsf{ReqOld}(2,1)$ and $\mathsf{ReqNew}(1)\,\mathsf{ReqOld}(2,1) + \mathsf{ReqNew}(2)\,\mathsf{ReqOld}(1,1)$ respectively. The detailed algorithm is shown in Fig.3.

Memory node —— A memory node has token memories $X_1^{\text{up}}, X_1^{\text{down}}, X_2^{\text{down}}, \ldots$ as explained before. Each memory node runs in either of two modes: *buffering mode* and *non-buffering mode*. When receiving a M_{feed} message, a non-buffering memory node sends the received data to all downward nodes immediately, while a buffering memory node does not send them until required by a M_{reqnew} message. Token memories are scanned when actions for M_{reqold} and M_{reqnew} messages are taken. The detailed algorithm is shown in Fig.4.

3 Features and Usage

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

$M_{\text{init}}(R)$: $R_1^{\text{up}} \leftarrow R,\ m_1^{\text{up}} \leftarrow [R, M_{\text{connect}}(R^{\text{self}}, 1)]$
$X_1^{\text{up}} \leftarrow \mathsf{ReqOld}(1,1)$
$X_i^{\text{down}} \leftarrow X_1^{\text{up}}$ (for all i)
M_{free} : $[R_1^{\text{up}}, M_{\text{discon}}(m_i^{\text{up}})]$
$M_{\text{connect}}(R, m)$: same as M_{connect} for a test node
$M_{\text{discon}}(i)$: same as M_{discon} for a test node
$M_{\text{feed}}(1, X)$: $X_1^{\text{up}} \leftarrow X_1^{\text{up}} + X$
$(X_i^{\text{down}} \leftarrow X_1^{\text{up}},\ \mathsf{Feed}(i)$ (for all i)
- in non-buffering mode)
$M_{\text{reqold}}(i, X)$: reply XX_i^{down}
$M_{\text{reqnew}}(i)$: $X_1^{\text{up}} \leftarrow X_1^{\text{up}} + \mathsf{ReqNew}(1)$
$X \leftarrow X_1^{\text{up}} - X_i^{\text{down}}$
$X_i^{\text{down}} \leftarrow X_1^{\text{up}}$
reply X

Figure 4: Memory node.

3.1 Data-Driven Network

Let us suppose there are relations $X_1^{\text{db}}, X_2^{\text{db}}, \ldots$ on $A_1, A_2, \ldots$, and $A \subset \bigcup A_i$. When a logical formula ψ is given, a *query by* ψ means to compute $X_\psi = \{x \in U^A \mid x \text{ satisfies } \psi;\ x|_{A_i} \in X_i^{\text{db}} \text{ holds for all } i\}$ from $X_1^{\text{db}}, X_2^{\text{db}}, \ldots$. If the formula ψ 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 ψ 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.:

$$\left.\begin{array}{l} \longrightarrow \langle 1|R_1^{\text{mem}}|1\rangle \longrightarrow \langle 1| \\ \longrightarrow \langle 1|R_2^{\text{mem}}|1\rangle \longrightarrow \langle 2| \end{array}\right\} R^{\text{join}}|m\rangle \longrightarrow$$

instead of two-input nodes (where R_1^{mem} and R_2^{mem} are non-buffering memory nodes, and R^{join} is a join node). The network will have several ports of entry nodes $\langle m_1^{\text{ent}}|R_1^{\text{ent}}, \langle m_2^{\text{ent}}|R_2^{\text{ent}}, \ldots$ that correspond to $X_1^{\text{db}}, X_2^{\text{db}}, \ldots$ respectively, and one exit node R^{exit} that corresponds to X_ψ. The next step is to add a buffering memory node R_*^{exit} to the exit node by sending $M_{\text{init}}(R^{\text{exit}})$ message to R_*^{exit}. The final step is to execute:

$$m_*^{\text{exit}} \leftarrow [R_*^{\text{exit}}, M_{\text{connect}}(R^{\text{dummy}}, m^{\text{dummy}})]$$

to allocate an upward port in R_*^{exit} (where R^{dummy} is a non-null dummy node and m^{dummy} is an arbitrary integer). As shown in Fig.5, $R_*^{\text{exit}}|m_*^{\text{exit}}\rangle$ is used to retrieve X_ψ.

Whenever a tuple is added to or deleted from a relation X_i^{db}, the memory in R_*^{exit} is recomputed and updated

Initialize :	$[R_i^{\text{cnt}}, M_{\text{feed}}(m_i^{\text{cnt}}, X_i^{\text{db}})]$ (for all i)
	$X_\psi \leftarrow [R_*^{\text{exit}}, M_{\text{reqold}}(m_*^{\text{exit}}, 1)]$
Add x to X_i^{db} :	$(R_i^{\text{cnt}}, M_{\text{feed}}(m_i^{\text{cnt}}, +x))$
Delete x from X_i^{db} :	$(R_i^{\text{cnt}}, M_{\text{feed}}(m_i^{\text{cnt}}, -x))$
Update X_ψ:	$X_\psi \leftarrow X_\psi + [R_*^{\text{exit}}, M_{\text{reqnew}}(m_*^{\text{exit}})]$

Figure 5: Usage of a data-driven network.

Initialize : $X_\psi \leftarrow [R_*^{\text{exit}}, M_{\text{reqold}}(m_*^{\text{exit}}, 1)]$

Figure 6: Usage of a data-driven network when $X_i^{\text{db}} = 0$.

automatically. The resulting relation X_ψ is simply updated by data transmission between R_*^{exit} to X_ψ. If X_ψ is implemented as a reference to X_1^{up} in R_*^{exit}, even this transmission will be unnecessary.

Here we present examples of data-driven OD-RETE networks. Suppose $X_1^{\text{db}}, X_2^{\text{db}}$ are relations on A_1, A_2, and $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ where $P_1(A_1), P_2(A_2)$ and$Q(A)$ are atomic formulas on A_1, A_2, and $A = A_1 \cup A_2$, respectively. In this case, a data-driven OD-RETE network for ψ will be created as in Fig.7. Here $R_{P_i}^{\text{test}}$ and R_Q^{test} are test nodes for $P_i(A_i)$ and $Q(A)$, $R_{P_1 \wedge P_2}^{\text{join}}$ is a join node, $R_{P_i}^{\text{mem}}$ is a non-buffering memory node, and R_ψ^{mem} $(= R_*^{\text{exit}})$ is a buffering memory node. Next suppose $\psi' \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q'(A)$. Since formulas ψ and ψ' have a common subformula, these two formulas can share a subnetwork for the common subformula, as shown in Fig.8.

In most cases where RETE networks are used, $X_i^{\text{db}} = 0$ holds for all i in the initial state. In such a situation, the initialization step is simplified as Fig.6.

3.2 Request-Driven Network

The design goal of the OD-RETE algorithm is to incorporate the request-driven query evaluation mechanism into the RETE algorithm, which is based on data-driven query evaluation. When tokens are fed to a request-driven RETE network, the network stores those tokens into memories in it and does no more processing until it is requested to evaluate queries.

Let $X_1^{\text{db}}, X_2^{\text{db}}, \ldots$ be relations on $A_1, A_2, \ldots$ and ψ be a logical formula on a subset of $\bigcup A_i$. A request-driven OD-RETE network for ψ can be constructed according to the ordinary RETE algorithm; In construction, test nodes are used instead of one-input nodes, join nodes are used instead of two-input nodes, and buffering memory nodes $R_1^{\text{db}}, R_2^{\text{db}}, \ldots$ are used as entry nodes corresponding to $X_1^{\text{db}}, X_2^{\text{db}}, \ldots$ respectively. The port to retrieve X_ψ, the result of the query, is allocated by:

$$m^{\text{exit}} \leftarrow [R^{\text{exit}}, M_{\text{connect}}(R^{\text{dummy}}, m^{\text{dummy}})]$$

Here, R^{exit} is the exit node of the network. Thus the port

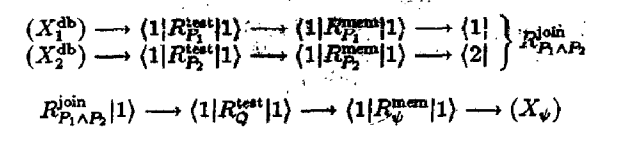

Figure 7: A data-driven network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$.

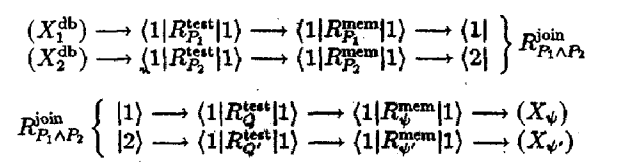

Figure 8: A data-driven network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ and $\psi' \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q'(A)$.

$R^{\text{exit}}|m^{\text{exit}}\rangle$ for X_ψ 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_1^{up} in R^{exit} to be updated. This happens in the initialization step and the updating step by M_{reqold}. If X_1^{up} in each R_i^{db} is implemented as a reference to a relation X_i^{db} 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 ways 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_1^{db} is a non-buffering memory node, and R_2^{db} and $R_{P_1 \wedge P_2}^{\text{mem}}$ are buffering memory nodes. Suppose X_2^{db} is a large relation in a DBMS. When a token $\pm x_1$ is fed to $\langle 1|R_1^{\text{db}}$, R_2^{db} receives $M_{\text{reqold}}(1, y)$ (where $y = x_1|_{A_1 \cap A_2}$) and will return yX_1^{down}. If X_1^{down} has an index whose key is $A_1 \cap A_2$, yX_1^{down} can be retrieved easily by utilizing the index. Thus, only required data in X_2^{db} 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

$$
\begin{array}{ll}
\text{Initialize :} & [R_i^{\text{db}}, M_{\text{feed}}(1, X_i^{\text{db}})] \text{ (for all } i) \\
& X_\psi \leftarrow [R^{\text{exit}}, M_{\text{reqold}}(m^{\text{exit}}, 1)] \\
\text{Add } x \text{ to } X_i : & (R_i^{\text{db}}, M_{\text{feed}}(1, +x)) \\
\text{Delete } x \text{ from } X_i : & (R_i^{\text{db}}, M_{\text{feed}}(1, -x)) \\
\text{Update } X_\psi : & X_\psi \leftarrow X_\psi + [R^{\text{exit}}, M_{\text{reqnew}}(m^{\text{exit}})]
\end{array}
$$

Figure 9: Usage of a request-driven network.

$$
\text{Initialize :} \quad X_\psi \leftarrow [R^{\text{exit}}, M_{\text{reqold}}(m^{\text{exit}}, 1)]
$$

Figure 10: Usage of a request-driven network when R_i^{ent} is implemented on a DBMS.

can grow (or shrink) even after they have been initialized and have been fed data.

For any node R in a OD-RETE network, it is possible to grow the network by sending $M_{\text{init}}(R)$ message to a new test/memory node S. Similarly, for any nodes R_1, R_2 in a OD-RETE network, it is possible to grow the network by sending $M_{\text{init}}(R_1, R_2)$ message to a new join node S. The newly created part of the network is initialized by M_{reqold} 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 $[R_{Q'}^{\text{test}}, M_{\text{init}}(R_{P_1 \wedge P_2}^{\text{join}})]$ and $[R_{\psi'}^{\text{mem}}, M_{\text{init}}(R_{Q'}^{\text{test}})]$. The newly created part is initialized by $X_{\psi'} \leftarrow [R_{\psi'}^{\text{mem}}, M_{\text{reqold}}(1, 1)]$.

A network can shrink by an M_{free} message. A node R is *removable* when R has no downward nodes (i.e., $R_i^{\text{down}} = 0$ for all i). The M_{free} 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 bases 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 integrities is an

$$
\begin{array}{l}
(X_1^{\text{db}}) \longrightarrow \langle 1|R_1^{\text{db}}|1\rangle \longrightarrow \langle 1|R_{P_1}^{\text{test}}|1\rangle \longrightarrow \langle 1| \\
(X_2^{\text{db}}) \longrightarrow \langle 1|R_2^{\text{db}}|1\rangle \longrightarrow \langle 1|R_{P_2}^{\text{test}}|1\rangle \longrightarrow \langle 2|
\end{array}
\Big\} R_{P_1 \wedge P_2}^{\text{join}}
$$

$$
R_{P_1 \wedge P_2}^{\text{join}} \begin{cases} |1\rangle \longrightarrow \langle 1|R_Q^{\text{test}}1|\rangle \longrightarrow (X_\psi) \\ |2\rangle \longrightarrow \langle 1|R_{Q'}^{\text{test}}|1\rangle \longrightarrow (X_{\psi'}) \end{cases}
$$

Figure 11: A request-driven network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ and $\psi' \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q'(A)$.

$$
\begin{array}{l}
(X_1^{\text{db}}) \longrightarrow \langle 1|R_1^{\text{db}}|1\rangle \longrightarrow \langle 1|R_{P_1}^{\text{test}}|1\rangle \longrightarrow \langle 1| \\
(X_2^{\text{db}}) \longrightarrow \langle 1|R_2^{\text{db}}|1\rangle \longrightarrow \langle 1|R_{P_2}^{\text{test}}|1\rangle \longrightarrow \langle 2|
\end{array}
\Big\} R_{P_1 \wedge P_2}^{\text{join}}
$$

$$
R_{P_1 \wedge P_2}^{\text{join}}|1\rangle \longrightarrow \langle 1|R_{P_1 \wedge P_2}^{\text{mem}} \begin{cases} |1\rangle \longrightarrow \langle 1|R_Q^{\text{test}}|1\rangle \longrightarrow (X_\psi) \\ |1\rangle \longrightarrow \langle 1|R_{Q'}^{\text{test}}|1\rangle \longrightarrow (X_{\psi'}) \end{cases}
$$

Figure 12: Another network for $\psi \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q(A)$ and $\psi' \equiv P_1(A_1) \wedge P_2(A_2) \wedge Q'(A)$.

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.

$$
\begin{array}{rl}
M_{\text{recfree}} : & \text{if there exists } i \text{ such that } R_i^{\text{down}} \neq 0 \\
& \text{then return immediately} \\
& (R_h^{\text{up}}, M_{\text{discon}}(m_h^{\text{up}})),\ (R_h^{\text{up}}, M_{\text{recfree}}) \\
& \text{(for all upward port } h)
\end{array}
$$

Figure 13: M_{recfree} message to recursively free nodes.

$$(X_1^{\mathrm{db}}) \longrightarrow \langle 1|R_1^{\mathrm{db}}|1\rangle \longrightarrow \langle 1|R_{P_1}^{\mathrm{test}}|1\rangle \longrightarrow \langle 1| \\ (X_2^{\mathrm{db}}) \longrightarrow \langle 1|R_2^{\mathrm{db}}|1\rangle \longrightarrow \langle 1|R_{P_2}^{\mathrm{test}}|1\rangle \longrightarrow \langle 2| \Big\} R_{P_1\wedge P_2}^{\mathrm{join}}$$

$$R_{P_1\wedge P_2}^{\mathrm{join}} \begin{cases} |1\rangle \longrightarrow \langle 1|R_Q^{\mathrm{test}}|1\rangle \longrightarrow \langle 1|R_2^{\mathrm{mem}}|1\rangle \longrightarrow (X_2^{\mathrm{view}}) \\ |2\rangle \longrightarrow \langle 1|R_Q^{\mathrm{test}}|1\rangle \longrightarrow \langle 1|R_3^{\mathrm{mem}}|1\rangle \longrightarrow (X_3^{\mathrm{view}}) \end{cases}$$

$$R_{P_i}^{\mathrm{test}}|2\rangle \longrightarrow \langle 1|R_{P_i'}^{\mathrm{test}}|1\rangle \longrightarrow \langle 1|R_1^{\mathrm{mem}}|1\rangle \longrightarrow (X_1^{\mathrm{view}})$$

Figure 14: A network for materialized views.

Initialize :	$[R_h^{\mathrm{db}}, M_{\mathrm{feed}}(1, X_h^{\mathrm{db}})]$
	$X_i^{\mathrm{view}} \leftarrow [R_i^{\mathrm{mem}}, M_{\mathrm{reqold}}(1,1)]$ (for $h = 1,2;\ i = 1,2,3$)
Add x to X_h^{db} :	$(R_h^{\mathrm{db}}, M_{\mathrm{feed}}(1, +x))$
Delete x from X_h^{db} :	$(R_h^{\mathrm{db}}, M_{\mathrm{feed}}(1, -x))$
Update X_i^{view}:	$X_i^{\mathrm{view}} \leftarrow X_i^{\mathrm{view}} + [R_i^{\mathrm{mem}}, M_{\mathrm{reqnew}}(1)]$
Remove X_i^{view}:	$[R_i^{\mathrm{mem}}, M_{\mathrm{discon}}(1)]$
	$[R_i^{\mathrm{mem}}, M_{\mathrm{recfree}}]$

Figure 15: Usage of the network for materialized views.

4.2 Materialized View

View materialization means to evaluate a view definition and store its results. The resulting relation 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 untolerable The problem addresses how to quickly update a view in response to database changes. This problem is different from the traditional view update problem. User can only update base relations and direct update to views are 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. The access to the materialized view is performed by sending M_{reqnew} to the memory node.

For example, Fig.14 is a network for multiple views. In this figure, X_h^{db} is a database relation. R_h^{db} is a non-buffering memory node. R_i^{mem} is a buffering memory node. The conditions corresponding to the views are as follows:

$$\begin{aligned} X_1^{\mathrm{view}} &: \psi_1(X_1^{\mathrm{db}}) \equiv P_1(X_1^{\mathrm{db}}) \wedge P_1'(X_1^{\mathrm{db}}) \\ X_2^{\mathrm{view}} &: \psi_2(X_1^{\mathrm{db}}, X_2^{\mathrm{db}}) \equiv P_1(X_1^{\mathrm{db}}) \wedge P_2(X_2^{\mathrm{db}}) \wedge Q(X_1^{\mathrm{db}}, X_2^{\mathrm{db}}) \\ X_3^{\mathrm{view}} &: \psi_3(X_1^{\mathrm{db}}, X_2^{\mathrm{db}}) \equiv P_1(X_1^{\mathrm{db}}) \wedge P_2(X_2^{\mathrm{db}}) \wedge Q'(X_1^{\mathrm{db}}, X_2^{\mathrm{db}}) \end{aligned}$$

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

When a database is updated, new tokens corresponding to the update flow into the network and views are differentially updated. 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 a DBMS. 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 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

[1] ADIBA, M. E., and LINDSAY, B. G., Database snapshots, *Proc. VLDB '80* (1980), 86–91.

[2] AJITOMI, N., An Enhanced RETE Algorithm and Its Formal Description, *RICOH Technical Report*, Vol.18 (1988), 24-27.

[3] ARAYA, S. et. al., Incremental Structuring of Knowledge Base, *Trans. of Inst. of Electronics, Information and Communication Engineers*, Vol. J71-D, No.6 (1988), 1100–1108.

[4] BLAKELEY, J. A., LARSON, P. A. and TOMPA, F. W., Efficiently updating materialized view, *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Vol.1986 (1986), 61–71.

[5] BROWNSTON, L., *Programming expert systems in OPS5*, Addison-Wesley (1985).

[6] CHAKRAVARTHY, U. S., and MINKER, J., Multiple Query Processing in Deductive Databases using Query Graphs, *Proc. VLDB '86* (1986), 384–391.

[7] FERNANDEZ, E. B., SUMMERS, R. C., and WOOD, C., *Database Security and Integrity*, Addison-Wesley (1981).

[8] FORGY, C. L., Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem, *Artificial Intelligence*, Vol. 19 (1982), 17–37.

[9] FORGY, C. L., *The OPS83 User's Manual*, Production Technologies, Inc. (1986).

[10] LANGLEY, P., Exploring the Space of Cognitive Architectures, *Behavior Research Method and Instrumentation*, Vol.15 (1983), 289–299.

[11] LINDSAY, B., HAAS, L., MOHAN, C., PIRAHESH, H. and WILMS, P., A snapshot differential refresh algorithm, *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Vol.1986 (1986), 53–60.

[12] MEDEIROS, C., and TOMPA, F. W., Understanding the implications of view update policies, *Proc. VLDB '85* (1985), 316–323.

[13] MIRANKER, D. P., TREAT: A Better Match Algorithm for AI Production Systems, *Proc. AAAI-87* (1987), 42–47.

[14] NICOLAS, J. M., Logic for Improving Integrity Checking in Relational Data Bases, *Acta Informatica*, Vol.18, No.3 (1982), 227–253.

[15] SCHOR, M. I., DALY, T. P., LEE, H. S., and TIBBITTS, B. R., Advances in RETE Pattern Matching, *Proc. Natl. Conf. Artif. Intell.*, Vol. 1 (1986), 226–232.

[16] SELLIS, T. K., Multiple Query Optimization, *ACM TODS*, Vol.13, No.1 (1988), 23–52.

[17] SHMUELI, O., and ITAI, A., Maintenance of views, *ACM SIGMOD Rec*, Vol.14, No.2 (1984), 240–255.

[18] SUBIETA, K., and RZECZKOWSKI, W., Query Optimization by stored queries, *VLDB '87* (1987), 369–380.

[19] TANO, S., MASUI, S., SAKAGUCHI, S., and FUNABASHI, M., A Fast Pattern Match Algorithm for A Knowledge Based Systems Building Tool - EUREKA, *Trans. on Info. Proc. Soc. of Japan*, Vol.28, No.12 (1987), 1255–1268.