ARGUS: Rete + DBMS = Efficient Continuous Profile Matching on Large-Volume Data Streams

Chun Jin, Jaime Carbonell
Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave. Pittsburgh, PA 15213
{cjin, jgc}@cs.cmu.edu

July 6, 2004

Abstract

Most existing data stream management projects focus on applications in which answers are generated at a fairly high rate. We identify an important sub-class of data stream applications, Stream Anomaly Monitoring Systems (SAMS), in which answers are generated very infrequently (i.e. the underlying SQL queries have extremely high selectivity), but may generate very high-urgency alerts when they do match. We exploit this property to produce significant optimization via an extension of the Rete algorithm, minimizing intermediate join sizes. A SAMS monitors transactions or other structured data streams and generates alerts when anomalies or potential hazards are detected. SAMS applications include but are not limited to potential financial crime monitoring, epidemiology tracking for first signs of new outbreaks, and potential fraud analysis on stock market transaction streams.

An important secondary objective is to build the SAMS on top of mature DBMS technology, to preserve full DBMS functionality and to permit full modularization of functions, guaranteeing cross-DBMS portability. The difficult part is to accomplish this objective while maximizing performance. But, the very-high-selectivity nature of SAMS applications permits us to achieve both objectives: efficiency and full DBMS compatibility.

We introduce the ARGUS prototype SAMS, which exploits very-high-selectivity via an extension of the Rete algorithm on a DBMS. The primary Rete extensions in our implementation are User-Defined Join Priority and Transitivity Inference. Preliminary experiments show the effectiveness and efficiency of the ARGUS approach over a standard Oracle DBMS. We further investigate several new techniques for Rete optimization: computation sharing, incremental aggregation, time window detection, and index selection.

Keywords: Stream Data, Continuous Query, Rete, Incremental Query Evaluation, Transitivity Inference, Database.

Acknowledgements: This work was funded by the Advanced Research and Development Activity as part of the Novel Intelligence from Massive Data program with contract NMA401-02-C-0033. The views and conclusions are those of the authors, not of the U.S. government or its agencies. We would like to thank Chris Olston for his helpful suggestions and comments, and Phil Hayes, Bob Frederking, Eugene Fink, Cenk Gazen, Dwight Dietrich, Ganesh Mani, Aaron Goldstein, and Johny Mathew for helpful discussions.

Contents

1 Introduction

As data processing and network infrastructure continue to grow rapidly, data stream processing quickly becomes possible, demanding, and prevalent. A Data Stream Management System (DSMS) [58] is designed to process continuous queries over data streams. Existing data stream management projects focus on applications in which answers are generated at a fairly high rate. We identify an important sub-class of data stream applications, Stream Anomaly Monitoring Systems (SAMS), in which answers are generated very infrequently (i.e. the underlying SQL queries have extremely high selectivity). We exploit this property to produce significant optimization via an extension of the Rete algorithm, minimizing intermediate join sizes. A SAMS monitors transaction data streams or other structured data streams and alert when anomalies or potential hazards are detected. The conditions of anomalies or potential hazards are formulated as continuous queries over data streams composed by experienced analysts. In a SAMS, data streams are composed of homogeneous records that record information of transactions or events. For example, a stream could be a stream of money transfer transaction records, stock trading transaction records, or in-hospital patient admission records. A SAMS is expected to periodically process thousands of continuous queries over rapidly growing streams at the daily volume of millions of records in a timely manner.

Examples motivating a SAMS can be found in many domains including banking, medicine, and stock trading. For instance, given a data stream of FedWire money transfers, an analyst may want to find linkages among big money transfer transactions connected to suspected people or organizations that may invite further investigation. Given data streams from all the hospitals in a region, a SAMS may help with early alerting of potential diseases or bio-terrorist events. In a stock trading domain, connections among suspiciously high profit trading transactions may draw an analyst's attention for further check whether insider information is illegally used. Comparing to traditional On-Line Transaction Processing (OLTP), a SAMS usually runs complex queries with joins and/or aggregations involving multiple facts and data records. Processing data streams dynamically and incrementally, a SAMS also distinguishes itself from On-Line Analytic Processing (OLAP).

It has been well recognized that many traditional DBMS techniques are useful for stream data processing. However, traditional DBMS's by themselves are not optimal for stream applications because of performance concerns. While a DSMS benefits from many well-understood DBMS techniques, such as operator implementation methods, indexing techniques, query optimization techniques, and buffer management strategies, a traditional DBMS is not designed for stream processing. A traditional DBMS is assumed to deal with rather stable data relations and volatile queries. In a stream application, queries are rather stable, persistent or residential, and data streams are rapidly changing. It is usual to expect a stream system to cope with thousands of continuous queries and high data rates. Therefore, while utilizing traditional DBMS techniques, many general-purpose stream projects, such as STREAM, TelegraphCQ, Aurora, and NiagaraCQ, are developing stream systems from scratch. However, for specific applications, such as SAMS's, it turns out that we can get away with an implementation on top of a DBMS.

ARGUS is a prototype SAMS that exploits the very-high-selectivity property with the adapted Rete algorithm, and is built upon the platform of a full-fledged traditional DBMS. For illustration purpose in this paper, we assume ARGUS runs in the FedWire Money Transfer domain. However, ARGUS is designed to be general enough to run with any streams. In ARGUS, a continuous query is translated into a procedural network of operators: selections, binary joins, and aggregations, which operates on streams and relations. Derived (intermediate) streams or relations are conditionally materialized as DBMS tables with old data truncated to maintain reasonable table sizes. A node of the network, or the operator, is represented as one or more simple SQL queries that perform the incremental evaluation. The whole network is wrapped as a DBMS stored procedure. Registering a continuous query in ARGUS involves two steps: create and initialize the intermediate tables, and install and compile the stored procedure.

In this paper, we first identify the unique characteristics of SAMS vs. general stream processing, and compare the streams with other types of streams, such as network traffic data, sensor data, and Internet data. Keeping the above observations and comparisons in mind, we then discuss alternative approaches to SAMS including those explored in other stream projects, and justify our choice of the ARGUS system design. We then present a detailed description of our Rete-based ARGUS design. We conclude with performance results and motivate future work.

2 Overview of Stream Anomaly Monitoring Systems

Figure 1 shows the SAMS dataflow. Data records in streams arrive continuously. They are matched against continuous queries registered in the system, and are stored in the database. Old stream data are truncated to keep the database manageable. Analysts selectively formulate the conditions of anomalies or potential hazards as continuous

queries, and register them with the SAMS. Registered queries are scheduled periodical executions over the data streams, and return new results (e.g. alerts), if any.

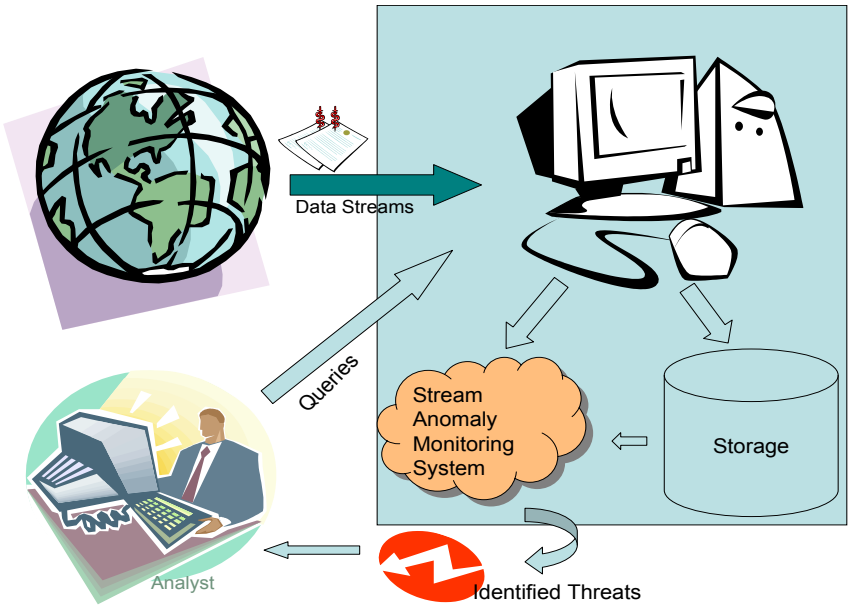

Figure 1: Data Flow of a Stream Anomaly Monitoring System

2.1 SAMS Characteristics

Formulating a query to capture anomalies or potential hazards buried in the data streams is a complex process. For many cases, an anomaly or hazard is too complex to be accurately represented as a query, and/or an accurate form is too expensive to evaluate. Therefore, an analyst usually formulates a continuous query that contains necessary but possibly not sufficient conditions of anomalies or hazards. The results of the continuous query invites the analyst's further investigation the scope of which may be far beyond the capability of a declarative or procedural language. For example, the analyst may search text information in libraries or on Internet to understand correlations between a disease and patients' geographies, or to know more about the relations between two organizations.

To formulate a continuous query, the analyst may start with a set of identified anomalies and hazards in historical data. He may use clustering tools to inspect and analyze the data, and induce/hypothesize necessary conditions of anomalies or hazards. To ensure that the induced conditions return reasonable results, the analyst may formulate the conditions as an ad hoc query (an ad hoc query is a one-time query compiled and executed against a Database snapshot), and run it against the historical data. Based on the returned results, the analyst may refine or rehypothesize the conditions. Several trial and error runs may be needed to decide the proper set of the query conditions. Therefore, a SAMS should support fast ad hoc query executions, and easy or automated conversion from an ad hoc query to a continuous query is preferable to reduce the analyst's query formulating effort.

Satisfaction of a continuous SAMS query is often rare, even if the query only tests necessary conditions of anomalies or hazards. It is usual that a query returns no new results in days. This is because a query result represents a potential anomaly or hazard, and by its very nature is rare. Also, to reduce postprocessing work by human, an analyst refines the query to return as few false positive results as possible, namely formulates the query with the maximum necessary conditions to filter out as many uninteresting data as possible. This characteristic distinguishes SAMS from many other stream applications that work with sensor data, network traffic data, or Internet data. By appropriately arranging the operators of the query plan, it is very likely to make every set of intermediate results very small. Materialization of intermediate results is therefore a feasible approach to incremental evaluation

and computation sharing.

Many stream projects developed extensions to SQL to support rich sets of window specifications, such as STREAM CQL [4] and TelegraphCQ's for-loop constructs [14]. Comparing to these extensions, SAMS uses only time-based sliding windows, which can be naturally formulated as query predicates over data attributes.

The purpose of introducing sliding windows is to convert the operations on unbounded streams to bounded data sets. For many stream applications, stream tuples are timestamped at the time they are generated, and the meaningful sliding windows are defined on the timestamp attribute. Due to various reasons, such as network delays and hardware failures, the arrival of the data tuples of an input stream may not restrictively conform to the timestamp order. Some stream systems relax the conformance to the data arrival order. The relaxation of the conformance is measured by a slackness parameter, which can be defined as a number n, or a fixed length of time T. When the slackness is defined as a number n, the system assumes that a tuple d generated at time τ will never arrive after more than n tuples that are later than τ arrive to the system. In other words, the system ignores a tuple d generated at time τ if the number of tuples later than τ that arrive before d exceeds the slackness parameter n. Slackness can also be defined as a fixed length of time T. After the system sees a tuple generated later than time $\tau + T$, the system assumes that no tuple generated at time τ or earlier will arrive in the future. The slackness parameter T is used in SAMS to decide when to truncate old stream data without fearing that any future data tuples may join or aggregate with the discarded data. Data truncated from the stream will not participate in continuous query evaluation any more, but a substantial portion or its totality may remain in the database for historical data analysis. It is worth noticing that the SAMS slackness is different from other systems' slackness notion, such as Aurora's slackness. Aurora's slackness is a query-level parameter that tells query executor what slack data to be ignored. SAMS slackness is a stream-level parameter. The value of SAMS slackness should conform to the stream characteristics. For example, assume a stock trading transaction is timestamped at its closing time and it must be reported to the system once it is closed, then setting the slackness $T = 2\ days$ may be sufficient for truncation purpose.

In terms of query semantics, a SAMS must support selection, join and aggregation. Support of selection and join over aggregated values is also needed. We call selection and join over aggregated values *post-aggregation*. Post-aggregation can be realized by defining views over aggregated values, and formulating the select-project-join query over the views. For some stream applications, such as network traffic analysis and sensor data applications, data aggregations are basic operations. In many cases, data items are first grouped and aggregated before filtering and joining. Post-aggregation is prevalent in such systems. This is due to the fact that data statistics are very important for decision making in those systems. The statistic feature of the data streams also justifies the employment of approximation techniques when the system is overloaded. However, this is not the case for SAMS streams.

We summarize the SAMS characteristics as following:

- A SAMS must support fast ad hoc query executions over a large volume of historical data.
- Easy or automated conversion from an ad hoc query to a continuous query is preferable.
- Matching results for continuous queries is not typically frequent.
- A SAMS uses only application or user settable time-based sliding windows.
- SAMS slackness is a stream-level parameter.
- A SAMS does not require the output to be in strict time order.
- A SAMS supports the following SQL query semantics: selection, join, aggregation, and post-aggregation.
- Aggregation and post-aggregation are not prevalent in a SAMS.
- Approximation techniques are not employed in a SAMS.
- Continuous queries may number in the thousands or tens of thousands.
- Daily stream volumes may exceed millions of records.

2.2 SAMS Query Examples

We choose the FedWire money transfer domain for illustration and experiments in this paper. For this particular application, there is a single data stream that contains money transfer transaction records, one record per transaction. A relevant subset of the attributes of the stream is shown below

TRANID	NUMBER(10),	– transaction id, the primary key
TYPE_CODE	NUMBER(4),	– transfer type
TRAN_DATE	DATE,	– transaction date
AMOUNT	NUMBER,	– transfer amount
SBANK_ABA	NUMBER(9),	– sending bank ABA number
SBANK_NAME	VARCHAR2(100),	– sending bank name
RBANK_ABA	NUMBER(9),	– receiving bank ABA number
RBANK_NAME	VARCHAR2(100),	– receiving bank name
ORIG_ACCOUNT	VARCHAR2(50),	– originator account
BENEF_ACCOUNT	VARCHAR2(50),	– beneficiary account

We present several continuous query examples, and formulate the queries in SQL language. Post-aggregation is realized by defining views (subqueries are more expressive in terms of specifying correlated queries).

Example 1 *The analyst is interested if there exists a bank, which received an incoming transaction over 1,000,000 dollars and has performed an outgoing transaction over 500,000 dollars on the same day.*

The query can be formulated as:

```
SELECT  r1.tranid rtranid, r2.tranid stranid,
        r1.rbank_name rbank_name,
        r1.tran_date rtran_date,
        r1.amount ramount, r2.amount smount
FROM    transaction r1, transaction r2
WHERE   r1.rbank_aba = r2.sbank_aba                    AND
        to_char(r1.tran_date, 'YYYYMMDD')
        = to_char(r2.tran_date, 'YYYYMMDD')            AND
        r1.amount > 1000000                            AND
        r2.amount > 500000;
```

Example 2 *For every big transaction, the analyst wants to check if the money stayed in the bank or left it within ten days.*

The query can be formulated as:

```
SELECT  r1.tranid rtranid, r2.tranid stranid,
        r1.rbank_name rbank_name,
        r1.benef_account benef_account,
        r1.tran_date rtran_date,
        r2.tran_date stran_date,
        r1.amount ramount, r2.amount smount
FROM    transaction r1, transaction r2
WHERE   r1.rbank_aba = r2.sbank_aba                    AND
        r1.benef_account = r2.orig_account             AND
        r1.tran_date <= r2.tran_date                   AND
        r1.tran_date + 10 >= r2.tran_date              AND
        r1.amount > 1000000                            AND
        r2.amount = r1.amount;
```

Example 3 *For every big transaction, the analyst again wants to check if the money stayed in the bank or left it within ten days. However he suspects that the receiver of the transaction would not send the whole sum further at*

once, but would rather split it into several smaller transactions. The following query generates an alert whenever the receiver of a large transaction (over $1,000,000) transfers at least half of the money further within ten days of this transaction.

The query can be formulated as:

$$
\begin{array}{lll}
\text{SELECT} & r.tranid\ tranid,\ r.rbank\_aba\ rbank\_aba, & \\
 & r.benef\_account\ benef\_account, & \\
 & AVG(r.amount)\ ramount, & \\
 & SUM(s.amount)\ samount & \\
\text{FROM} & transaction\ r,\ transaction\ s & \\
\text{WHERE} & r.rbank\_aba = s.sbank\_aba & AND \\
 & r.benef\_account = s.orig\_account & AND \\
 & r.tran\_date <= s.tran\_date & AND \\
 & s.tran\_date <= r.tran\_date + 10 & AND \\
 & r.amount > 1000000 & \\
\text{GROUP BY} & r.tranid,\ r.rbank\_aba,\ r.benef\_account & \\
\text{HAVING} & SUM(s.amount) > AVG(r.amount) * 0.5; &
\end{array}
$$

Example 4 *Suppose for every big transaction of type code 1000, the analyst wants to check if the money stayed in the bank or left within ten days. An additional sign of possible fraud is that transactions involve at least one intermediate bank. The query generates an alert whenever the receiver of a large transaction (over $1,000,000) transfers at least half of the money further within ten days of this transaction using an intermediate bank.*

The query can be formulated as:

$$
\begin{array}{lll}
\text{SELECT} & r1.sbank\_aba\ sbank,\ r1.orig\_account\ saccount, & \\
 & r1.rbank\_aba\ rbank,\ r1.benef\_account\ raccount, & \\
 & r1.amount\ ramount,\ r1.tran\_date\ rdate, & \\
 & r2.rbank\_aba\ ibank,\ r2.benef\_account\ iaccount, & \\
 & r2.amount\ iamount,\ r2.tran\_date\ idate, & \\
 & r3.rbank\_aba\ frbank,\ r3.benef\_account\ fraccount, & \\
 & r3.amount\ framount,\ r3.tran\_date\ frdate & \\
\text{FROM} & transaction\ r1,\ transaction\ r2,\ transaction\ r3 & \\
\text{WHERE} & r2.type\_code = 1000 & AND \\
 & r3.type\_code = 1000 & AND \\
 & r1.type\_code = 1000 & AND \\
 & r1.amount > 1000000 & AND \\
 & r1.rbank\_aba = r2.sbank\_aba & AND \\
 & r1.benef\_account = r2.orig\_account & AND \\
 & r2.amount > 0.5 * r1.amount & AND \\
 & r1.tran\_date <= r2.tran\_date & AND \\
 & r2.tran\_date <= r1.tran\_date + 10 & AND \\
 & r2.rbank\_aba = r3.sbank\_aba & AND \\
 & r2.benef\_account = r3.orig\_account & AND \\
 & r2.amount = r3.amount & AND \\
 & r2.tran\_date <= r3.tran\_date & AND \\
 & r3.tran\_date <= r2.tran\_date + 10; &
\end{array}
$$

Example 5 *Check whether there is a bank, having incoming transactions for more than $100,000,000 and outgoing transactions for more than $50,000,000 on one particular day.*

The formulated query is composed of three SQL statements. The first two are view definitions, and the last one is the query on the views.

```
CREATE VIEW   rbank_money AS
SELECT        r1.rbank_aba rbank_aba,
              r1.tran_date tran_date,
              SUM(r1.amount) rsum
FROM          transaction r1
GROUP BY      r1.rbank_aba, r1.tran_date
HAVING        SUM(r1.amount) > 100000000;
```

```
CREATE VIEW   sbank_money AS
SELECT        r2.sbank_aba sbank_aba,
              r2.tran_date tran_date,
              SUM(r2.amount) ssum
FROM          transaction r2
GROUP BY      r2.sbank_aba, r2.tran_date
HAVING        SUM(r2.amount) > 50000000;
```

```
SELECT   r.rbank_aba rbank_aba,
         s.sbank_aba sbank_aba,
         r.tran_date tran_date,
         r.rsum rsum,
         s.ssum ssum
FROM     rbank_money r, sbank_money s
WHERE    r.rbank_aba = s.sbank_aba          AND
         r.tran_date = s.tran_date;
```

Example 6 *Get the transactions of Citibank and Fleet in a period of time.*

The query can be formulated as:

```
SELECT   r1.tranid tranid,
         r1.sbank_name sbank_name,
         r1.tran_date tran_date,
         r1.amount amount
FROM     transaction r1
WHERE    (r1.sbank_name =' Citibank (New York State)'          OR
         r1.sbank_name =' Fleet Bank')                         AND
         r1.tran_date >= to_date('20021120', 'YYYYMMDD')       AND
         r1.tran_date <= to_date('20021130', 'YYYYMMDD');
```

Example 7 *The analyst is interested whether Citibank has conducted a transaction on a particular day with the amount exceeding 1,000,000 dollars.*

The query can be formulated as:

```
SELECT   r1.tranid tranid,
         r1.sbank_name sbank_name,
         r1.tran_date tran_date,
         r1.amount amount
FROM     transaction r1
WHERE    r1.sbank_name =' Citibank (New York State)'          AND
         to_char(r1.tran_date, 'YYYYMMDD') =' 20021126'       AND
         r1.amount > 1000000;
```

3 Designs of Stream Anomaly Monitoring Systems

Many stream systems use an extension to SQL as their query language, such as STREAM CQL [4], and TelegraphCQ's extension of SQL [14] with for-loop constructs. The purpose of the extensions is to support rich window specifications over streams. Some projects, such as Aurora, use a procedural query language. Aurora's query language is a GUI tool to specify a query as a network of connected operator boxes.

A stream system may also face the problem of resource management and plan scheduling. Depending on implementation, some stream systems simplify and decouple resource management, scheduling, and query execution. ARGUS is such an example. Resource management in ARGUS is handled by the underlying DBMS and the scheduling is simplified as pushing stream data through the shared query plan network.

3.1 Alternative Approaches to SAMS

In terms of implementation methodology, there are two distinctive approaches, DBMS-based and workflow-oriented. We compare the stream systems in this classification dimension.

Workflow-oriented approaches employ procedural query languages to specify continuous queries over streams. Aurora [1][12] is such an example. An Aurora query is specified as a network of connected operator boxes over streams. Without compilation and optimization, the network represents an execution plan ready to be executed. Aurora performs optimization on the network by rearranging subnetworks when it is necessary. With such an approach, a user has extensive control over the query execution logic, and may create optimized network for multiple queries (exploiting computation sharing to the extent possible). This approach requires users to have very detailed knowledge of the system and to devote considerable manual effort. When there are many queries, a user is less likely to create an optimized network, and the optimization must be left to the system.

A DBMS-based approach uses a declarative query language, usually an extension to SQL, to specify continuous queries over streams. Such an approach allows a user to focus only on query semantics, and not to worry about how a query is executed. Although a user may manually modify an execution plan, the practice is not common. It is up to the DSMS to optimize the query execution plan, and share computation among multiple queries.

A DSMS developed with DBMS-based approach looks like an extension to a DBMS, and employs many DBMS techniques. How the DBMS techniques are applied in a DSMS distinguishes two sub-approaches of this methodology. One sub-approach uses well-understood DBMS techniques at the source code level. This sub-approach may either extend and adapt a DBMS code base to stream processing architecture, such as TelegraphCQ [14][46], or implementing the whole system from scratch including well-known DBMS algorithms, such as STREAM [58][7]. The other sub-approach uses full-fledged DBMS as a platform upon which a stream processing module is built. This approach decouples optimization and execution between the underlying DBMS and the stream processing module. The platform sub-approach is employed by two earlier systems, Alert [67] and Tapestry [78], and also by ARGUS.

The clear advantage of the platform approach is the reusability of well-honed DBMS systems and their transportability across platforms. Many processing issues are handled by the underlying DBMS, such as access path selection, join methods, and resource management, etc. The platform approach leads to fast implementations with possible loss of some flexibility. The platform approach may not be a good general choice, but for SAMS, it is a very logical and appropriate one.

For a SAMS, the platform approach provides ready-made powerful ad hoc query support over historical data, and a baseline stream system for comparison. SQL, the ad hoc query language, can also serve as the continuous query language. Thus, an ad hoc query, tested and refined by analysts over historical data, can be registered as a continuous query without modification. Many DBMS's provide procedural language extensions to SQL for handling complex logics [26][66]. Such advances allow incremental evaluation schemes, such as Rete [27], to be implemented inside a database.

With various query rewriting and optimization techniques, many rarely-matched queries can be compiled to execution plans with small intermediate result tables. SAMS's assumption of rare matching justifies the choice of intermediate result materialization. No intensive requirement on Quality of Service for each query and requirement of no approximation justify the decoupling of incremental execution and sharing from resource management and scheduling. Resource management is indirectly controlled by optimization of the incremental execution scheme and sharing to minimize intermediate result sizes.

Alert [67] and Tapestry [78] are two other platform-based systems. Alert targets an active database, and uses triggers to check the query conditions, and modified cursors to fetch satisfied tuples. This method may not be

efficient to handle high data rates and the large number of queries in a stream processing scenario. Tapestry is closer to ARGUS. Tapestry's incremental evaluation scheme is also wrapped in a stored procedure. However, its incremental evaluation is realized by rewriting the query with sliding window specifications on the append-only relations (streams). A sliding window specifies the period of time from the last time that the query is executed to the current time. Data tuples accumulated during this period of time are the new ones that need to be processed. This approach becomes inefficient when the append-only table is very large. Although adding an index on the timestamp attribute to the table can speed up the query execution significantly, maintenance overhead of the index on the fast changing tables may be too high to make the performance acceptable. Also, the nature of the rewritten query makes the computation sharing among multiple queries very difficult.

3.2 Adapted Rete Algorithm

Figure 2 shows the ARGUS system architecture. The new data elements are appended to database tables. The continuous queries are converted to Rete networks with ReteGenerator, and installed in the database. The Rete networks run periodically on new data arrivals, store and update intermediate results, and generate alerts when any continuous query matches the new data.

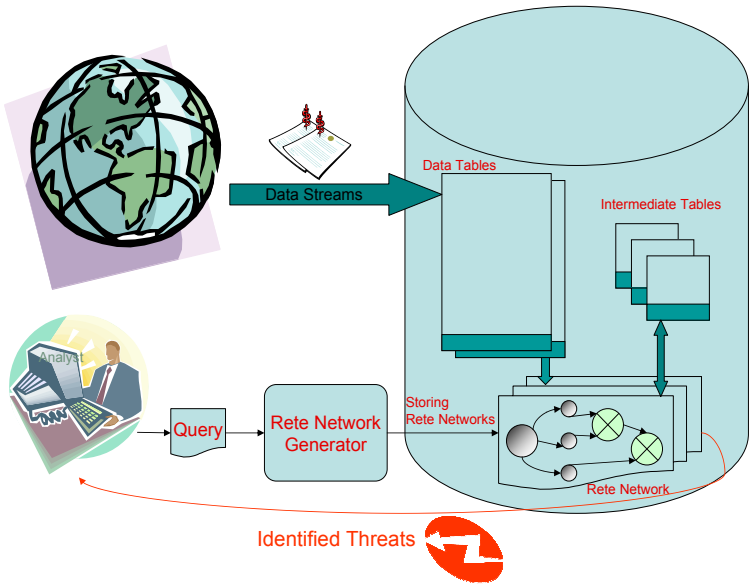

Figure 2: The Architecture of the ARGUS Profile System

The Rete match algorithm was developed by Charles L. Forgy [27] for use in production system interpreters. It is an efficient method for matching a large collection of patterns to a large collection of objects. Because of its efficiency, many commercial rule-based production systems use Rete for pattern matching. Rete can be adopted for incremental query evaluation. A Rete network can be viewed as a tree that data streams flow through and join on the internal nodes. TREAT [55] is similar to Rete except that it joins the data streams all together at one node, and thereby omits the internal join nodes. Readers are referred to [27] for details on the Rete algorithm. Here, we demonstrate how we adapt Rete algorithm to incremental query evaluation.

Let n and m denote the old data sets, and Δn and Δm the new much smaller incremental data sets, respectively. By Relational Algebra, a selection operation σ on data $n + \Delta n$ is equivalent to $\sigma(n + \Delta n) = \sigma(n) + \sigma(\Delta n)$. $\sigma(n)$ is the set of old results that is materialized. To evaluate incrementally, only the computation on Δn portion is needed $(\sigma(\Delta n))$. Similarly, for a join operation $\bowtie$ on $(n + \Delta n)$ and $(m + \Delta m)$, we have $(n + \Delta n) \bowtie (m + \Delta m) = n \bowtie m + \Delta n \bowtie m + n \bowtie \Delta m + \Delta n \bowtie \Delta m$. $n \bowtie m$ is the set of old results that is materialized. Only the computations on

$\Delta n \bowtie m + n \bowtie \Delta m + \Delta n \bowtie \Delta m$ portion are needed, which can be decomposed to three joins. When Δn and Δm are small compared to n and m, the time complexity of the incremental join is linear to $O(n + m)$.

Consider Example 4 in Section 2. A satisfied result set contains joins of three tuples. The first tuple is the originating transaction that the original sender transferred at least $1,000,000 to an intermediate account. The second is an intermediate transaction in which the money is transferred from the intermediate account to the destination account. And the third is a transaction in which at least half of the money is distributed to another account. Each tuple in the result set has to satisfy a set of intra-tuple features (selection predicates). They are Pattern 1:

$$\begin{array}{lcll} r1.type\_code & = & 1000 & AND \\ r1.amount & > & 1000000 & \end{array}$$

Pattern 2:

$$r2.type\_code = 1000$$

and Pattern 3:

$$r3.type\_code = 1000$$

respectively.

Other conditions are inter-tuple features, or join predicates. Particularly, following are the inter-tuple features that join the first two tuples:

$$\begin{array}{lcll} r1.rbank\_aba & = & r2.sbank\_aba & AND \\ r1.benef\_account & = & r2.orig\_account & AND \\ r2.amount & > & 0.5 * r1.amount & AND \\ r1.tran\_date & \leq & r2.tran\_date & AND \\ r2.tran\_date & \leq & r1.tran\_date + 10 & \end{array}$$

and the following are that join the last two tuples:

$$\begin{array}{lcll} r2.rbank\_aba & = & r3.sbank\_aba & AND \\ r2.benef\_account & = & r3.orig\_account & AND \\ r2.amount & = & r3.amount & AND \\ r2.tran\_date & \leq & r3.tran\_date & AND \\ r3.tran\_date & \leq & r2.tran\_date + 10 & \end{array}$$

Figure 3 shows a Rete network of the query. In this Rete network, data tuples that pass Pattern 1 and Pattern 2 are joined first, then the results are joined with data tuples that pass Pattern 3. Joining Pattern 2 and Pattern 3, then Pattern 1 is also logically correct. However, the different orders of joins could affect performance dramatically as observed with traditional query optimization techniques.

Figure 4 shows the dataflow of a possible logical plan for the query which could be generated by a conventional DBMS query compiler. Figure 5 shows the dataflow of the Rete network. The logical plan and the Rete network have similar data access paths. However, in the Rete network, only necessary computations on the new data are carried out. The intermediate results are materialized in the database tables.

Now we use a small data set shown in Table 1 to illustrate how the Rete network works as new data continuously arrive. Suppose that the data records (tuples) arrive in the order of the *tranid*. At Time 0, there are 2 records, and the Rete network is in the status shown in Figure 6. Note that if no new data satisfy at a certain node, then none of its succeeding nodes needs to be evaluated. In this case, no data satisfies the node joining Pattern 1 and Pattern 2, so its succeeding nodes does not need to be evaluated. At Time 1, two new records arrive. The Rete network is updated as in Figure 7. At Time 2, one more new record arrives, as shown in Figure 8, it triggers the generation of an alert.

A continuous query may contain multiple SQL statements and a single SQL statement may contain unions of multiple SQL terms. Multiple SQL statements allow an analyst to define views. Each SQL term is mapped to a sub-Rete network. These sub-Rete networks are then connected to form the statement-level sub-networks. And the statement-level sub-networks are further connected based on the view references to form the final query-level Rete network.

In summary, a Rete network performs incremental query evaluation over the delta part (new stream data) and materializes intermediate results. The incremental evaluation makes the execution much faster. However, there

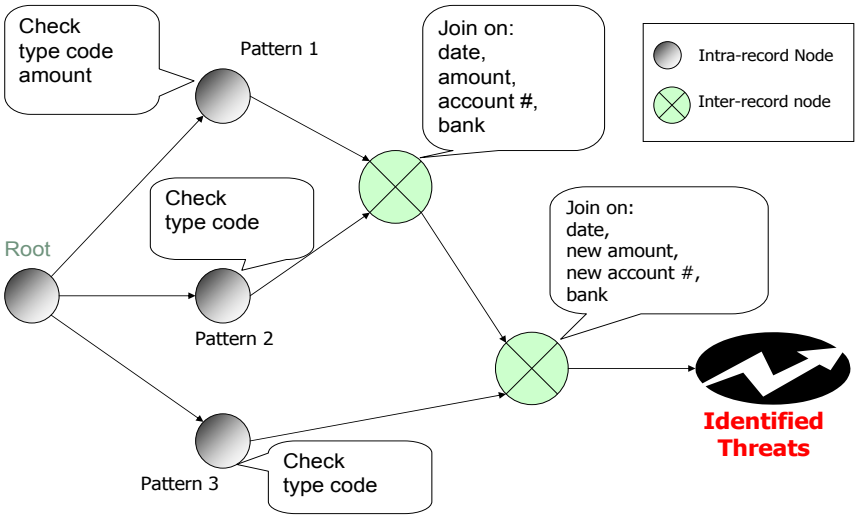

Figure 3: A Rete Network for Example 4

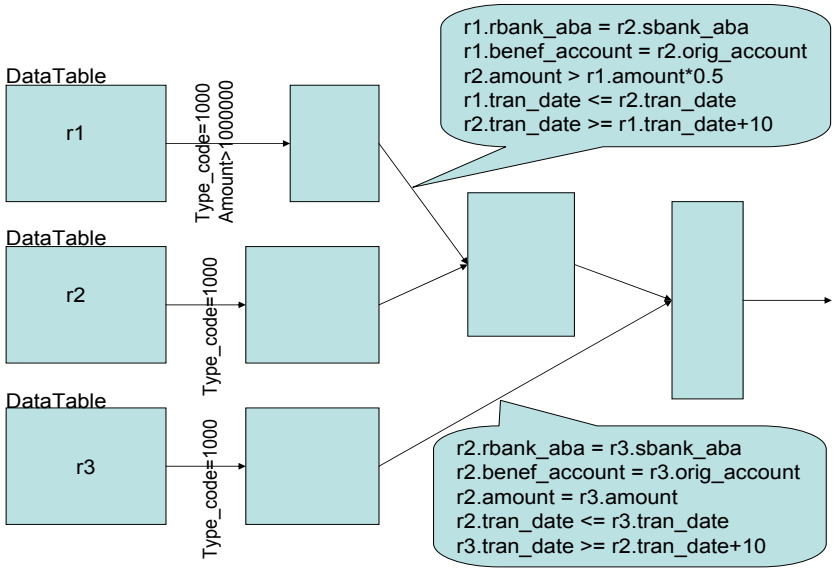

Figure 4: An Sample Logical Query Execution Plan for Example 4

is a potential problem when any materialized intermediate table is very large, which deteriorates the incremental evaluation performance severely. Only when the intermediate tables are fairly small, can the incremental evaluation scheme show consistent big advantages. Fortunately, since queries are not expected to be satisfied frequently, there

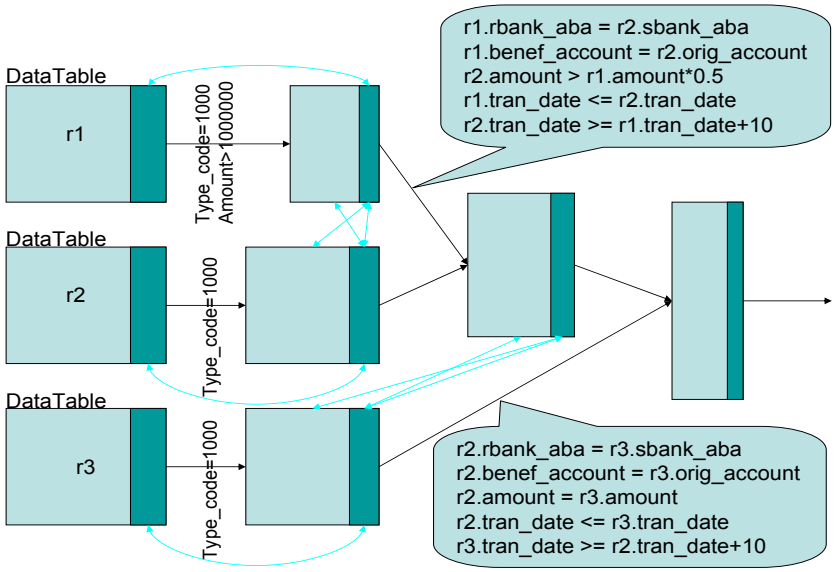

Figure 5: The Data Flow of the Rete Network for Example 4

tran id	type code	tran date	amount	sbank name	rbank name	orig account	benef account
1	1000	12/01/02	400,000	PNC	Fleet	1000001	1000009
2	1000	12/02/02	1,200,000	PNC	Citibank	3000001	2000001
3	1000	12/03/02	305,000	Citibank	Fleet	3000001	2000009
4	1000	12/05/02	800,000	Citibank	Chase	2000001	4000001
5	1000	12/06/02	800,000	Chase	Citizen's	4000001	5000009

Table 1: A Small Data Set to show how the Rete Network works

are usually highly selective conditions that make the intermediate tables fairly small. We investigated several effective ways of reducing the sizes of the intermediate tables. We will also discuss the cases in which intermediate tables can not be reduced to small sizes.

4 ARGUS Profile System Design

As shown in Figure 2, the ARGUS Profile System contains two components. One is the database created on the Oracle DBMS, and the other is the Rete construction module, ReteGenerator. A profile is translated into a Rete network by ReteGenerator. Then the Rete network is registered in the database. The registration includes creating the intermediate tables, initializing the tables based the historical data, and storing and compiling the Rete network wrapped in the stored procedure. The database schedules periodical runs of the active Rete networks against new data.

4.1 Database Design

A simplified ARGUS database schema is shown in Figure 9. New data arrives continuously and is appended to the data table. *Do_queries*() is a system level procedure that finds all the active queries from QueryTable in the order of their priorities, and executes them one by one. If any query is satisfied, an alert is generated. The job of running the procedure *Do_queries*() is scheduled periodically. QueryTable records query information, one entry per query. Each

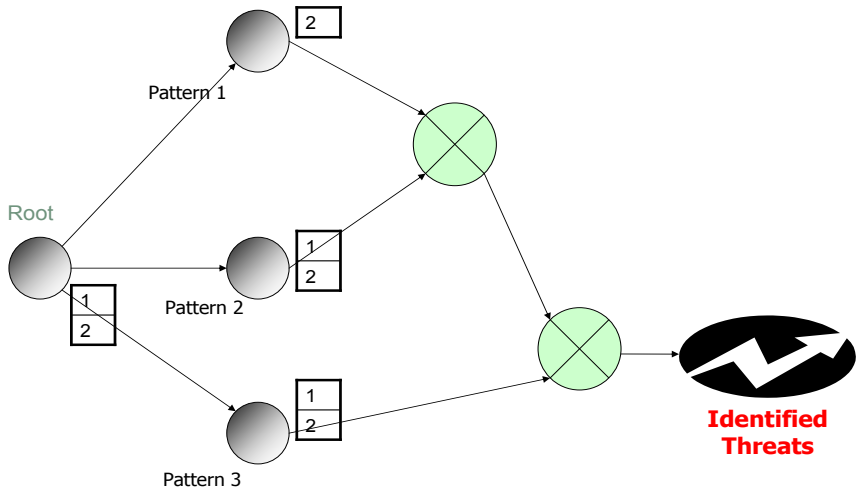

Figure 6: The Status of the Rete Network for Example 4 at Time 0. Records in the intermediate tables are represented by their *tranid*s.

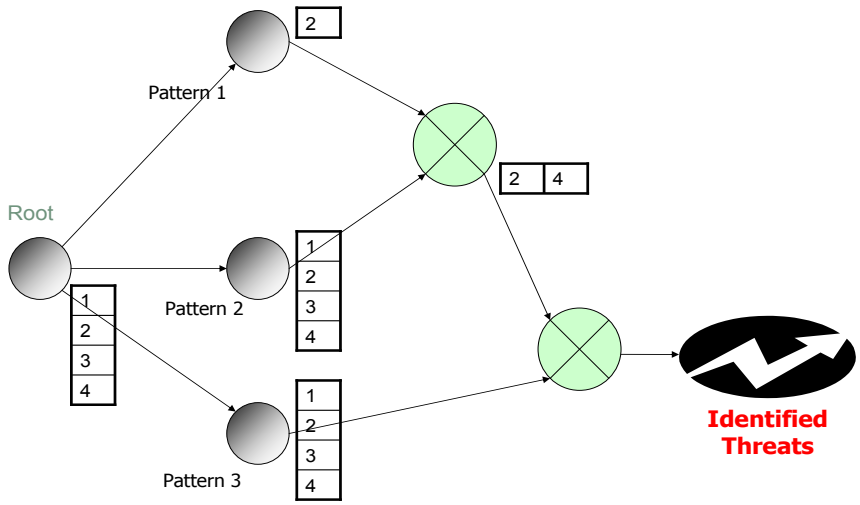

Figure 7: The Status of the Rete Network for Example 4 at Time 1. Records in the intermediate tables are represented by their *tranid*s.

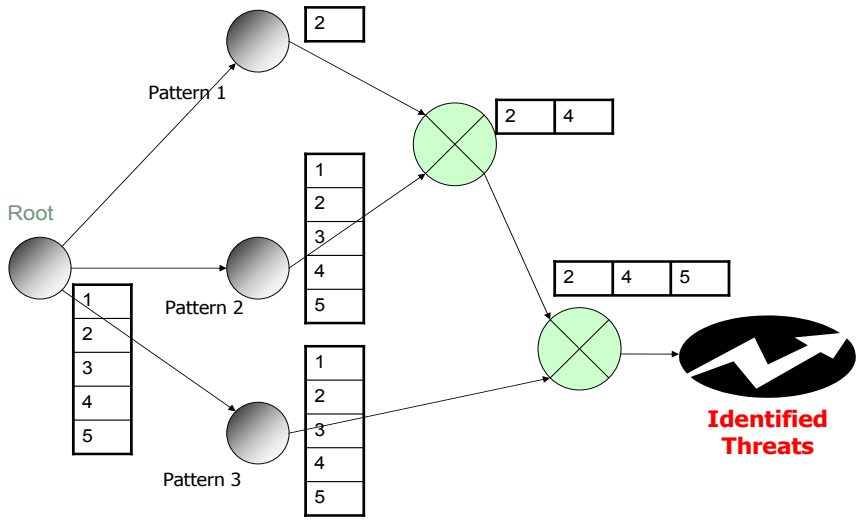

Figure 8: The Status of the Rete Network for Example 4 at Time 2. Records in the intermediate tables are represented by their *tranid*s.

entry contains the query ID, the procedure name to call, the query priority, and a boolean flag indicating whether the query is active or not. Only active queries are executed in a scheduled run. The query priorities set the order in which the query procedures are called. For example, a procedure with priority 1 precedes a procedure with priority 10.

4.2 Restrictions on the SQL

To apply the Rete algorithm, we impose some restrictions to the SQL queries. Therefore, the set of queries that the Rete networks support is a subset of the standard SQL language. Nevertheless, the subset is rich enough for a SAMS.

The first restriction is that the selection conditions of an SQL statement should be in a relaxed Disjunctive Normal Form. This means that the disjunctive selections should be specified by UNION unless the disjunctive predicates operate on a single table. For example, for the following SQL:

$$\begin{array}{ll} \text{SELECT} & r1.a \\ \text{FROM} & table1\ r1,\ table2\ r2 \\ \text{WHERE} & (r1.a < r2.b \text{ OR } r1.a > r2.c) \quad AND \\ & (r1.d < 1 \text{ OR } r1.e > 2); \end{array}$$

The Rete algorithm expects the input of the equivalent form:

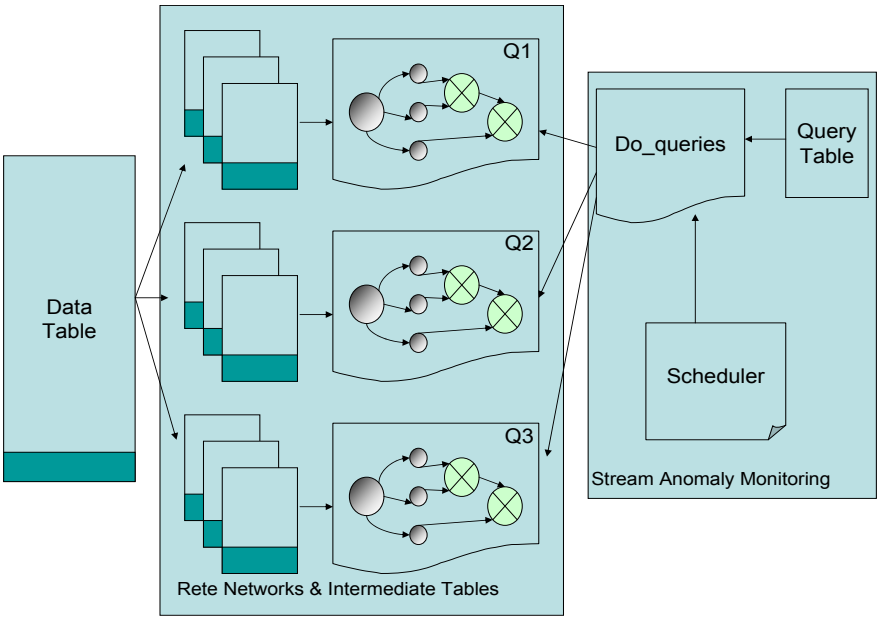

Figure 9: ARGUS Database Design

$$
\begin{array}{lll}
\text{SELECT} & r1.a & \\
\text{FROM} & table1\ r1,\ table2\ r2 & \\
\text{WHERE} & r1.a < r2.b & AND \\
 & (r1.d < 1 \text{ OR } r1.e > 2) & \\
\text{UNION} & & \\
\text{SELECT} & r1.a & \\
\text{FROM} & table1\ r1,\ table2\ r2 & \\
\text{WHERE} & r1.a > r2.c & AND \\
 & (r1.d < 1 \text{ OR } r1.e > 2); &
\end{array}
$$

Note that the disjunctive form of $(r1.d < 1$ OR $r1.e > 2)$ is allowed in an SQL because all the conditions in the form involve only one table. However, the other disjunctive form, $(r1.a < r2.b$ OR $r1.a > r2.c)$, is expected to be realized with UNION of two SQLs because the conditions in the form involve multiple tables.

For the Rete algorithm, selection predicates on a single table, even with disjunctions, are easily evaluated in one step. However, disjunctive selection predicates over multiple tables are not easy. By pushing up disjunctions to the top level, we restrict each sub-Rete network in a conjunctive form, and results are unioned in the last step.

The second restriction is that the current system doesn't support multiple-way joins. It will reject an SQL statement with purely multiple-way joins, which is rare in real applications. For example, the following multiple-way join statement is not accepted by the ReteGenerator:

$$
\begin{array}{ll}
\text{SELECT} & r1.a \\
\text{FROM} & table1\ r1,\ table2\ r2,\ table3\ r3 \\
\text{WHERE} & r1.a + r2.b = r3.c;
\end{array}
$$

But it accepts the following equivalent statement:

SELECT	$r1.a$	
FROM	$table1\ r1,\ table2\ r2,\ table3\ r3$	
WHERE	$r1.e = r2.e$	AND
	$r2.f = r3.f$	AND
	$r1.a + r2.b = r3.c;$	

4.3 Translating SQL Queries into Rete Networks

A query formulated by an analyst is a set of SQL statements. The corresponding Rete network is a database stored procedure translated by ReteGenerator. Appendix A shows a sample translation of the query in Example 4 into a Rete network wrapped in an Oracle stored procedure.

To simulate the topological connections of the Rete network, each node is associated with a Rete Flag, and maintains a list of Rete Flags of its children nodes (children flag list). A Rete Flag indicates whether the node needs to be updated or not. The default value of a Rete Flag is false. It is set to true only if any children flag is true which indicates that the update to the associated table is necessary. If none of children Rete Flags is true, then the node doesn't need to be updated.

To install a Rete network, we need both the procedure and a set of auxiliary DDL statements that create the intermediate tables and initialize the tables to contain the results on historical data. The initialization is necessary when the old data is present. Appendix B shows the corresponding DDL statements for the Rete network of Example 4.

ReteGenerator contains three components: the SQL Parser, the Rete Topology Constructor, and the Rete Coder. The SQL Parser parses a set of SQLs to a set of parse trees, the Rete Topology Constructor rearranges the connections of nodes in a parse tree to obtain the desired Rete network topology, and the Rete Coder traverses the reconstructed tree, and generates the Rete network code in Oracle PL/SQL language.

The SQL Parser takes the query (a set of SQLs) as input, and outputs a set of parse trees. Figure 10 shows the structure and the data flow of the parser. It contains an SQL parsing module and a lexer module. The SQL parsing module is a Perl package generated by a compiler compiler Perl-byacc [20] based on an SQL grammar [48]. The lexer module is also a Perl package [82]. The lexer is called by the SQL parsing module to tokenize the input query. The stream of tokens is fed to the SQL parsing module to generate the parse trees. The tools we used, namely, Perl-byacc, Perl Lexer, and the SQL grammar, were all from open sources, and were modified to meet our special needs.

4.3.1 Rete Topology Construction

Rete Topology Constructor constructs the network topology based on the selection predicates and join predicates. Only the *where_clause* subtree decides the Rete Topology. Figure 11 shows the schematic subtree of the query in Example 4. Note that this subtree is pretty flat with all conditions on the same level, which is purposely done during parsing for easy reconstruction of the Rete network.

Rete Topology Constructor, a Perl function, goes through the following three steps to construct the Rete topology.

First, predicates are classified based on the tables they use. For the subtree in Figure 11, the classifications are shown in Figure 12.

Second, the classified sets of the predicates are sorted based on the number of tables that the sets contain:

$[r1,\ r2,\ r3,\ (r1,\ r2),\ (r2,\ r3)]$

Each set corresponds to a node in the reconstructed subtree. From now on, we will use the term *node* to refer a classified set of predicates.

Finally, the new subtree is reconstructed bottom-up. We first look for a pair of single-table nodes for the merge step. A pair of single-table nodes can be merged if there is a joint-table node joining just these two tables. Merge continues until all nodes are merged into a single node. Figure 13 shows the reconstructed subtree.

4.3.2 Aggregation and Union

An SQL statement may contain groupby/having clauses. If it also contains a *where_clause*, the Rete network is generated for the *where_clause* and the output of the Rete network is stored in a table, which will be the input to

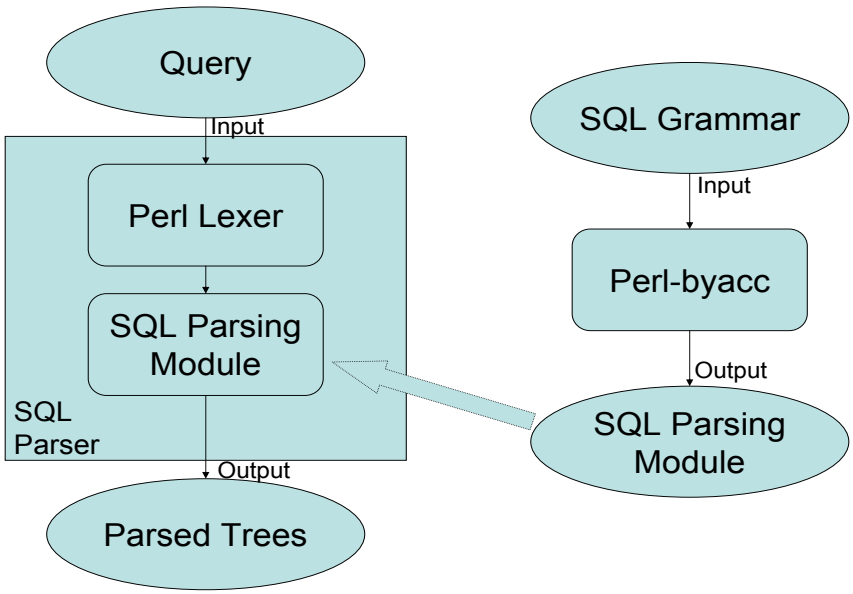

Figure 10: Building the SQL Parser

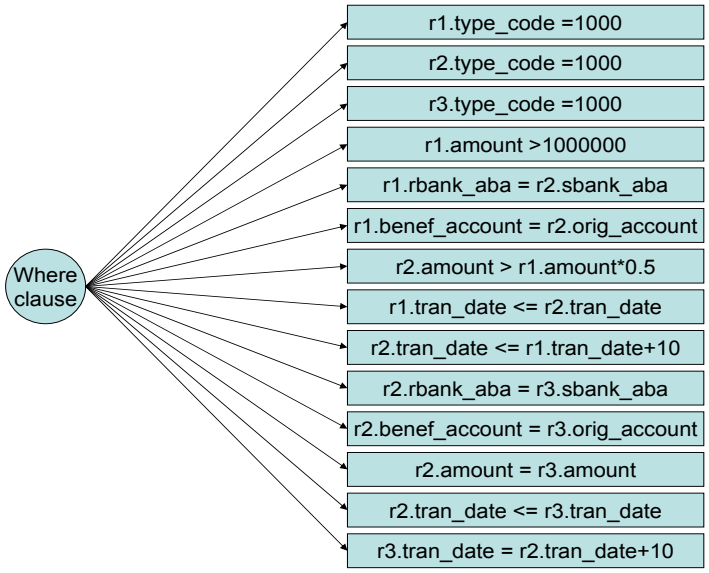

Figure 11: The *where_clause* parse tree for Example 4

groupby/having clauses. We tried two methods for handling groupby/having clauses. The first method, implemented in our system, does the operations of grouping and having always on the whole input table, and finds the difference between the current results and the previous results. The difference is considered as the incremental results. The

$r1$:

$r1.type\_code = 1000$
$r1.amount > 1000000$

$r2$:

$r2.type\_code = 1000$

$r3$:

$r3.type\_code = 1000$

$r1,\ r2$:

$r1.rbank\_aba = r2.sbank\_aba$
$r1.benef\_account = r2.orig\_account$
$r2.amount > 0.5 * r1.amount$
$r1.tran\_date <= r2.tran\_date$
$r2.tran\_date <= r1.tran\_date + 10$

$r2,\ r3$:

$r2.rbank\_aba = r3.sbank\_aba$
$r2.benef\_account = r3.orig\_account$
$r2.amount = r3.amount$
$r2.tran\_date <= r3.tran\_date$
$r3.tran\_date <= r2.tran\_date + 10$

Figure 12: Condition Classifications for Example 4

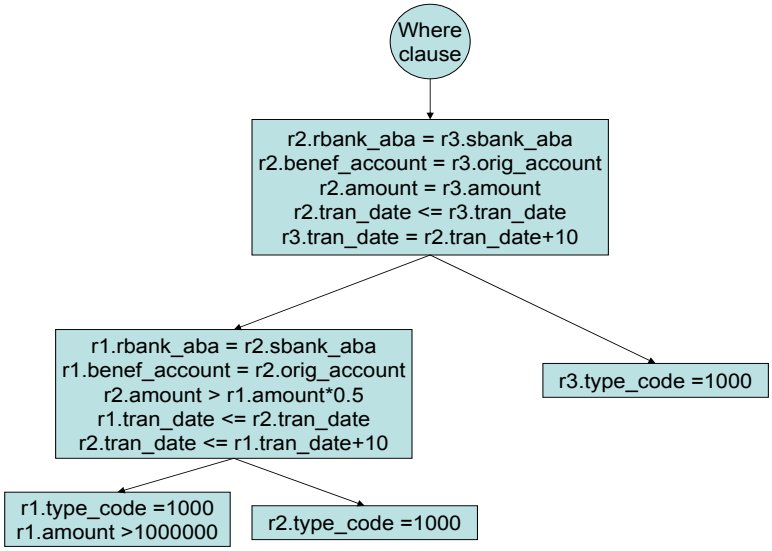

Figure 13: The reconstructed *where_clause* parse tree for Example 4

second method extracts the records whose groups need updates from the input table, regroups these records with new data records, and applies the having conditions. The rationale for the second method is that by grouping only the groups that are changed, we save the computations on unchanged groups. However, experiments show that the second method is rather slow, probably because a large portion of groups need to be updated in our data sets. The overhead introduced are not effectively compensated by the computation reduction.

A query may contain multiple SQL statements and a single SQL statement may contain unions of multiple SQL terms. When there are no group/having clauses in any term of a query, a Rete network is constructed bottom-up as usual. If there are groupby/having clauses somewhere in a UNION term or in a view, Rete sub-networks are constructed from all the branches bottom-up until a groupby clause is encountered. The presence of the *groupby_clause* voids the incremental evaluation. The first *groupby_clause* in each branch is a split point. The operations preceding the split point are handled incrementally by Rete networks as usual, and the operations succeeding the split point are always handled on whole inputs, and no materialized results are maintained.

Example 3 and Example 5 in Section 2.2 demonstrate the two different places of groupby/having clauses in queries. Appendix C shows the generated procedure code for Example 5. Because the views only have groupby/having clauses and no *where_clause*, there is no Rete network generated for this query. The procedure code is almost the same as the original query except that the code for detecting differences between the new results and the old results are generated. For this special case, we don't have a way to use Rete. The query in Example 3 contains a *where_clause*, so a Rete network is generated for it. The output of the Rete network is the input to the groupby/having clauses. Appendix D shows the generated procedural code.

5 Improvements on Rete Network

Various optimization techniques can be used to minimize intermediate result sizes. We have tried and integrated two methods into the current ReteGenerator: User-Defined Join Priority, and Transitivity Inference. Our experiments show that each can have significant effects on performance.

5.1 User-Defined Join Priority

Join priority specifies, among all possible join orders, which should be followed. This is very similar to the reordering of join operators in traditional query optimization. To see why this priority makes difference, let's again look at Example 4. Assume we have 100000 records, $type\_code = 1000$ is a non-selective condition with selectivity factor of 90%(90% of the records meet the condition), and $amount > 1,000,000$ is a selective condition with selectivity factor of 0.1%. We also assume *type_code* is independent of *amount*. Therefore, the numbers of the records in the three intermediate tables corresponding to the three sets of single-table selection predicates are $100000 * 0.9 * 0.001 = 9$, $100000 * 0.9 = 90000$, and $100000 * 0.9 = 90000$, respectively. According to the joint-conditions, we can either merge the first two single-table nodes, or the last two. We expect much less intermediate results for merging the first two, and we should do this merge first. However, without the knowledge of the table statistics, it is hard for the ReteGenerator to choose the best join order, and thus is left to the analyst. The ReteGenerator accepts user-defined join priority. It assumes that the order of the tables appearing in the $from\_clause$ of an SQL is the order of joining. Applying query optimization techniques based on cost models for Rete network construction is left as one piece of our future work.

5.2 Transitivity Inference

Transitivity Inference explores the transitivity property of comparison operators, such as $>$, $<$, and $=$, to infer hidden selective single-table conditions from a set of existing conditions. For illustration, let's again look at Example 4. We have the following conditions:

$$\begin{array}{lcll} r1.amount & > & 1000000 & and \\ r2.amount & > & r1.amount * 0.5 & and \\ r3.amount & = & r2.amount & \end{array}$$

$r1.amount > 1000000$ is very selective. Actually, by the transitivity property of operator $>$, $r1.amount > 1000000$ and $r2.amount > r1.amount * 0.5$ imply a selective condition on $r2$: $r2.amount > 500000$.

Further, with the condition $r3.amount = r2.amount$, another selective condition on $r3$ can be derived: $r3.amount > 500000$.

These inferred conditions have significant impact on performance. The first level intermediate tables, filtered by the highly selective single-table selection predicates, are made very small, which saves many computations on subsequent joins. Section 6 gives details on the performance analysis with respect to Transitivity Inference.

Currently, we implemented a simple version of the transitivity inference in the Rete Topology Constructor. Among many derivable cases, the current module only derives several special yet commonly-seen cases. Given a pair of conditions, if one condition contains exactly one column a, which we call Single-Table Condition with Single-Column (STCSC), and the other contains exactly two columns a, and b from two different tables, which we call Joint-Table Conditions with Single-Columns (JTCSC), then the inference module will try to infer some predicate on b. Just for convenience, in the following text, STCSC and JTCSC also refer to the parse trees of the conditions.

The inference module is comprised of two parts. The first part builds up the data structures from the existing conditions. The second part loops through the data structures to look for the hidden conditions. Figure 14 shows the data structures and work flow of the inference module.

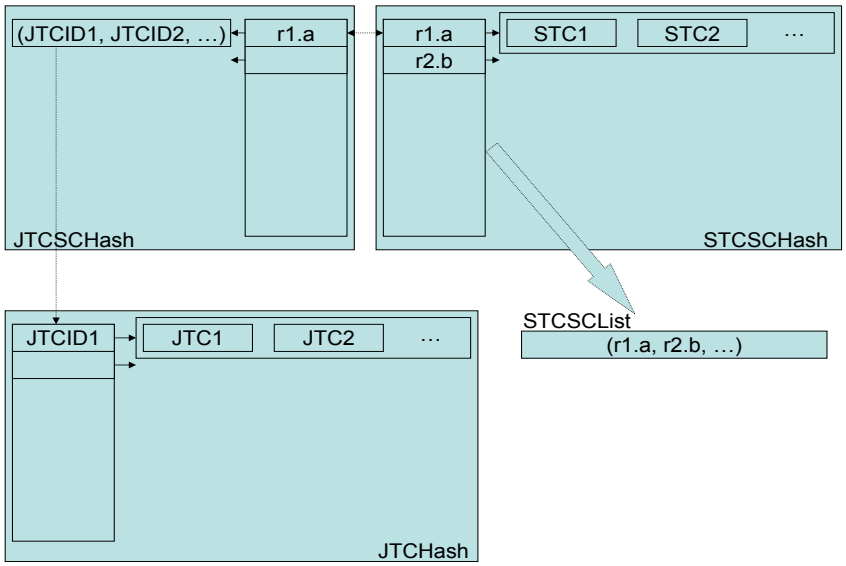

Figure 14: Data Structures for Transitivity Inference

JTCHash is a hash table that stores all the JTCSCs. The key of JTCHash is a numerical ID, and the value is a JTCSC. STCSCHash is a hash table that stores all the STCSC. The key of the STCSCHash is a column name, and the value is the list of STCSCs that contain the key column. JTCSCHash is a hash table that stores all the JTCSCs. The key of the JTCSCHash is a column name, and the value is the list of IDs of JTCSCs that contains the key column. STCSCList is the list containing all the keys of STCSCHash.

Following is the inference algorithm.

1. Pop up a column $r1.a$ from the STCSCList.
2. If there is an entry $r1.a$ in JTCSCHash, we do transitivity inference on each pair of the matching conditions: JTC1 and STC1.

   If a hidden condition C on column $r2.b$ is inferred from JTC1 and STC1,

   – Add C to the *where_clause* parse tree.

- Add C to STCSCHash, and add $r2.b$ to STCSCList. This allows inferred conditions to serve for further inference.
- Remove JTC1 from JTCHash. This step avoids endless cycle of inferring the same conditions. If we can't find the entry for a given a JTCSC ID in JTCHash, then we simply skip this JTCSC.

3. Repeat 1. and 2. until STCSCList is empty.

A transform function is developed to facilitate inference. Given a pair of STCSC STC1, and JTCSC JTC1, the function transforms the conditions into the following format:

$$\begin{aligned} \text{STC1:} \quad & r1.a >=< f_1(c) \\ \text{JTC1:} \quad & r1.a >=< f_2(r2.b) \end{aligned}$$

where $f_1(c)$ is a function of constant c, and $f_2(r2.b)$ is a function of $r2.b$. $>=<$ is the comparison operator, which could be one of the following: $<$, $\leq$, $>$, $\geq$, $=$. Based on the operator, a hidden condition on $r2.b$ may or may not be inferred.

6 Experimental Results

6.1 Experiment Setting

The experiments were conducted on the prototype ARGUS on an HP computer with 1.7G CPU and 512M RAM, running Windows XP.

6.1.1 Data Sets

The data sets are derived from a database of synthesized FedWire money transfer transactions. The database D contains 320006 records. The timestamps of the data spans 3 days.

For the experiments, we split the data in two ways, and the most of the experiments were conducted on both data conditions:

- Data Condition 1 (Data1). Old data: the first 300000 records of D. New data: the remaining 20006 records of D. This data condition provides alerts for most of the queries on testing.
- Data Condition 2 (Data2). Old data: the first 300000 records of D. New data: the next 20000 records of D. This data condition doesn't generate alerts for most of the queries on testing.

6.1.2 Queries

We used the seven queries described in Section 2.2 for the experiments. To test the effect of Transitivity Inference, we also ran queries of Example 2 and Example 4 in variant forms. Query variants with respect to Transitivity Inference differ with each other in two orthogonal dimensions. First, the original SQL query may or may not contain the manually added hidden conditions. Second, the Rete network may be generated with or without the Transitivity Inference module turned on.

Table 2 summarizes the features and settings of each query or query variant used in the experiments. Each query or query variant is assigned a unique query ID. Q1-Q7 are for the queries of the seven examples in a common setting: no hidden condition is added to the original queries, and Transitivity Inference module is turned on. Q8-Q11 are the variants of Example 2 and Example 4. Q8 and Q10 are variants of Example 2 and Example 4 whose Rete networks are generated without Transitivity Inference. Q9 is the variant of Example 4 whose original SQL query is enhanced with hidden conditions. Q11 is a variant of Example 4 whose Rete network is slightly different. This variant will be described later with respect to Partial Rete generation.

For each query, we run the original SQL query and the Rete network on the two data conditions. When running the original SQL queries, the data is the combination of the old data part and the new data part. Note that Rete networks need initialization which takes some time. However, the initialization is a one-time operation. Rete networks provide incremental new results, while original SQL queries only provide whole sets of results.

QueryID	Example	Having	CondAdd	UseTI	Diff
Q1	1	N	N	Y	N
Q2	2	N	N	Y	Y
Q3	3	Y	N	Y	N
Q4	4	N	N	Y	Y
Q5	5	Y	N	Y	N
Q6	6	N	N	Y	N
Q7	7	N	N	Y	N
Q8	2	N	N	N	Y
Q9	4	N	Y	Y	Y
Q10	4	N	N	N	Y
Q11	4	N	N	N	Y

Table 2: Experimental Queries. "Having": whether the query involves any aggregation functions. "CondAdd": whether hidden conditions that can be inferred by Transitivity Inference module are manually added to the original SQL query or not. "UseTI": whether the Rete network is generated with the Transitivity Inference module turned on. "Diff": whether there are hidden conditions that the Transitivity Inference may infer for the query. If there are, then we may expect performance difference between two Rete networks generated from a same query with one network uses Transitivity Inference and the other not. "Having" and "Diff" reflect the properties of a query. "CondAdd" and "UseTI" reflect settings for Rete network generation.

6.2 Results Interpretation

Table 3 summarizes the results of running the queries on the two data conditions. To be clearer, relevant information in Table 3 is also depicted in charts when we present detailed interpretations in later subsections.

QueryId	Rete Execution Time(s) Data1	SQL Execution Time(s) Data1	Alerts Data1	Rete Execution Time(s) Data2	SQL Execution Time(s) Data2	Alerts Data2
Q1	1	14	Y	1	12	Y
Q2	1	20	Y	1	19	N
Q3	16	20	Y	16	19	N
Q4	3	45	Y	3	31	N
Q5	14	14	Y	15	17	Y
Q6	1	6	N	1	7	N
Q7	1	6	Y	1	6	N
Q8	18	20	Y	17	19	N
Q9	2	20	Y	2	12	N
Q10	44	45	Y	27	31	N
Q11	38	45	Y	17	31	N

Table 3: Execution Times of the Queries on Data1 and Data2. "Rete Execution Time(s)": the time in seconds to run the Rete network of the query on the specified data condition. "SQL Execution Time(s)": the time in seconds to run the original SQL query on the specified data condition. "Alerts": whether the running of the query on the specified data conditions generates an alert or not.

Figure 15 and Figure 16 show the results in charts for Q1-Q7. For most of the queries, Rete networks with Transitivity Inference gain significant improvements over directly running the SQL queries.

6.2.1 Aggregation

Q3 and Q5 are the two queries involving aggregations. Although they both have groupby/having clauses, they are slightly different to each other. Q3 has a *where_clause* before the grouping operator, while Q5 doesn't have any *where_clause* before any grouping operator. Therefore, Q3 has a Rete network, but Q5 doesn't.

Q5 is a post-aggregation query. We can not apply Rete on such queries for incremental evaluations. In our system, the generated procedure reevaluates the query on the whole data set, and then finds the difference between

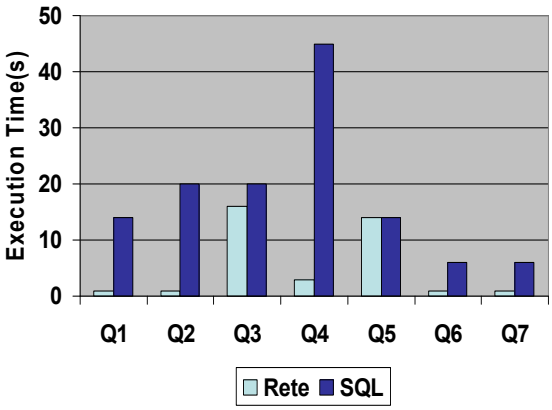

Figure 15: Execution Times of Q1-Q7 on Data1

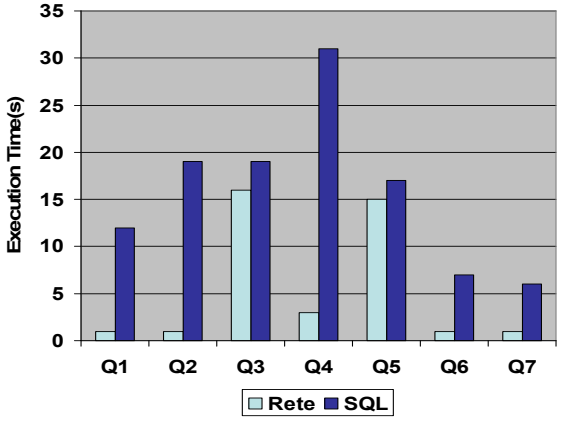

Figure 16: Execution Times of Q1-Q7 on Data2

the whole new results and the previous results. As shown in Figure 15 and Figure 16, it is not surprising that the Rete version (no actual Rete network inside) of Q5 doesn't gain improvement over the SQL query.

For Q3, the *where_clause* is mapped to a Rete network. The output table of the network is the input to the subsequent grouping operator. As shown in Figure 15 and Figure 16, the Rete network of Q3 does slightly better than the SQL query. It is important to note that the *where_clause* specifies a join on a small table (after selection) and a large table (the original data table). This operation is the major time consumer in the Rete network because the join involves a large table. Here the benefits of incremental evaluation on only small incremental parts are not achieved.

6.2.2 Transitivity Inference

Example 2 and Example 4 are the two queries that benefit from Transitivity Inference. Figure 17 and Figure 18 show the execution times for these two examples respectively. In the legends of the figures,

- Rete TI stands for the Rete network generated with Transitivity Inference,
- Rete Non-TI stands for the Rete network generated without Transitivity Inference,
- SQL Non-TI stands for the original SQL query,
- and SQL TI stands for the original SQL query with hidden conditions manually added.

The inferred condition $amount > 500000$ is very selective with selectivity factor of 0.1%. Clearly, when Transitivity Inference is applicable and the inferred conditions are selective, a Rete network runs much faster than its non-TI counterpart and the original SQL.

It is interesting to note in Figure 18 that *SQL TI* ($Q9$), the SQL query of Example 4 with manually added conditions, runs significantly faster than the original one. This means that transitivity inference is not applied in the DBMS query optimization, and actually can be a potential query optimization method for traditional DBMS queries.

6.2.3 Partial Rete Generation

In previous subsections, we see that the execution time of a Rete network without Transitivity Inference may be close to the original SQL query (Q8 and Q10). There are queries that Transitivity Inference is not applicable and single-table selection predicates for some of the join tables are not selective or do not exist at all. For such queries, we have to do joins with large tables.

In ARGUS, if single-table selection predicates are not present, the Rete network will not materialize the entire stream table, which significantly saves both space and execution time. However, when there are any single-table

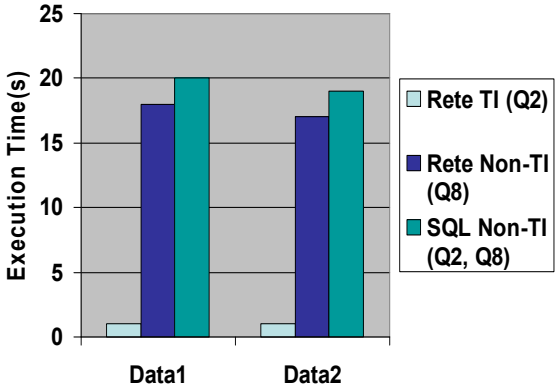

Figure 17: Effect of Transitivity Inference. Query execution times of Example 2 on Data1 and Data2.

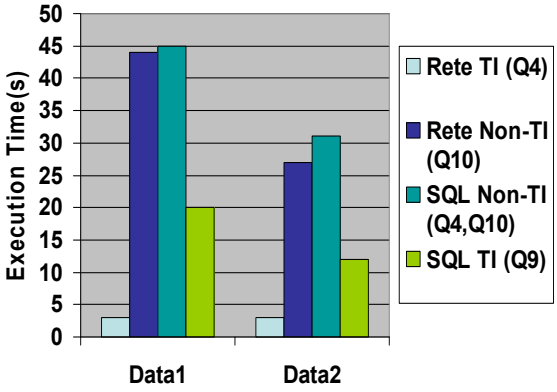

Figure 18: Effect of Transitivity Inference. Query execution times of Example 4 on Data1 and Data2.

selection predicates present, even highly non-selective ones, the Rete network will materialize the selection results. Because materialization on large intermediate results requires many I/O operations, it deteriorates the performance of a Rete work significantly. In such cases, pipelined operation is preferable than materialization.

Assume Transitivity Inference is not applicable by turning the module off, Q10 of Example 4 is such a query. The two single-table selection predicates ($r2.type\_code = 1000$, $r3.type\_code = 1000$) on two of the three join tables are highly non-selective, the sizes of the intermediate results are close to that of the original data table. As shown in Figure 18, the performance gain by the Rete network is very limited.

If ReteGenerator is aware of table statistics, such a materialization can be skipped. We tested a Rete network (Q11) with the two materializations replaced by the internal pipeline operation provided by the DBMS. We call this Partial Rete Generation. As this technique is not automatically implemented yet, the Rete network is generated by generating a Rete network with the SQL query that the non-selective conditions are removed, so corresponding intermediate results are not materialized, then the non-selective conditions are added manually back to the generated Rete codes.

Figure 19 shows the execution times of the Partial Rete network, the Rete network, and the original SQL. It is clear that in case of non-selective conditions present, Partial Rete is superior to the original Rete network.

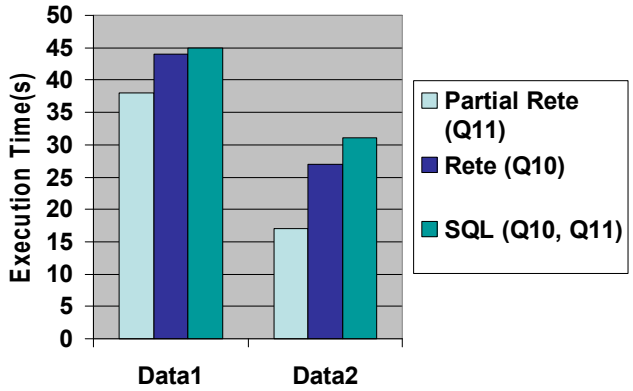

Figure 19: Effect of Partial Rete Generation. Comparing the execution times of Partial Rete, Non-Partial Rete, and SQL for Example 4 on Data1 and Data2.

6.2.4 Using Indexes

All the experiments discussed so far were conducted WITHOUT any secondary indexes on the original stream data table. Indexing can speed up the execution of equal or range predicates on the tables of large cardinalities. However, it also imposes heavy maintenance overheads [9][60][47] upon frequent updates to the stream table which is essential for a SAMS.

To study the effect of indexing on ARGUS, we conducted experiments with two indexes created on the stream table. One was created on a pair of attributes: ($rbank\_aba$, $benef\_account$), and the other on another pair: ($sbank\_aba$, $orig\_account$). The choice of the indexes is based on the observation that indexing attributes are the join attributes of the queries in our experiments.

Here we discuss the results of running queries Q3 and Q4 over Data1. Table 4 shows the execution times of running Q3 and Q4 over Data1 with Rete and SQL under two conditions: index is present and index is not present. Figure 20 shows the same results in charts. When indexes are present, both queries' SQLs are executed mush faster. But the two Rete networks response differently. The performance of Q3 Rete network is boosted significantly with the indexes, and Q4 Rete network doesn't gain any benefits from the indexes. The different responses of the Rete networks of Q3 and Q4 to the indexes become clear when we recall the following. Q3 doesn't have selective single-table selection predicates for one of the join tables, so the Rete network has to join with the large original data table. When the table is equipped with the indexes on join attributes, the Rete network takes advantage of the indexes. For Q4, with Transitivity Inference, each join table is filtered with highly selective single-table selection predicates and is very small.

It is worth mentioning the index maintenance overhead. When there are the indexes, inserting 20006 records (the new data part of Data1) takes about five minutes. When there is no index, inserting the same amount of data takes 20 seconds. Using indexes is a tradeoff balancing the benefits of joining with large tables and the cost of the index maintenance. It should be an empirical decision. A guideline is that if there are many queries whose Rete networks have to join with large original tables, so that the benefits exceed the maintenance cost, we should create appropriate indexes.

QueryID	Rete Index(s)	SQL Index(s)	Rete Non-Index(s)	SQL Non-Index(s)
Q3	16	19	2	10
Q4	3	31	3	14

Table 4: Comparing Index and Non-Index. Showing Execution Times in Seconds of Running Query Q3 and Q4 on Data1 under conditions: 1) Rete with Index, 2) SQL with Index, 3) Rete without Index, and 4) SQL without Index.

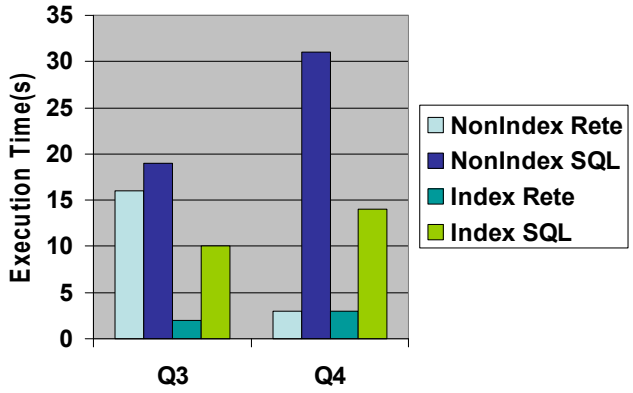

Figure 20: Comparing Index and Non-Index on Data1. Rete and SQL of Query Q3 and Q4 were run against Data1 under two conditions: indexes present or not.

7 Related Work

Besides Rete [27], and TREAT [56], the Match Box algorithm[62], and the LEAPS algorithm[57] are the two other algorithms used for pattern matching in production systems.

Gator networks [37] are applied in the Ariel database system to speed up condition testing for multi-table triggers. Gator is a generalization of the Rete and TREAT algorithms. It is a non-binary bushy tree, while Rete is a binary bushy tree (All Rete networks shown in [37] are left-deep trees, which is not necessarily the case in ARGUS). [37] shows that optimized Gator networks normally have a shape which is neither pure Rete nor pure TREAT, but an intermediate form where some but not all possible inner joins (β nodes) are materialized. In the subsequent work [38], Gator is applied into a scalable trigger processing system, in which predicate indexing is explored to provide computation sharing among common predicates and efficient detection for necessary computations upon data changes. The work on Gator networks is more general than our work with respect to employing non-binary discrimination networks with cost model optimizations. However, [37] explores only single-tuple update a time, doesn't consider aggregation operators, and is used for trigger condition detection instead of stream processing.

Traditional query optimization techniques [30], particularly the techniques that search the optimal execution plans [68][41] based on cost models [63][40], provide valuable insights to continuous query processing.

The Stanford STREAM project focuses on developing a general-purpose Data Stream Management System (DSMS) [7][58][8][6] that supports a declarative query language and can cope with high data rates and thousands of continuous queries. To exploit well-understood DBMS techniques, as well as new stream processing techniques for a general-purpose DSMS, to the extent possible, the STREAM group investigates various aspects of stream processing, particularly, 1) the stream query language, CQL, which is an extension of SQL to support stream semantics, 2)structures of query plans, accounting for plan sharing and approximation techniques, 3) resource management, and operator scheduling, 4) approximation techniques aiming at graceful performance degradation under limited resources, and 5) constraint exploiting to reduce memory overhead. ARGUS doesn't have explicit resource management and operator scheduling modules, whose functionalities are realized by the underlying DBMS system. However, ARGUS exploits techniques on query optimization, plan sharing, and data stream constraints to reduce resource requirement, particularly memory requirement. ARGUS's query language is SQL, which is sufficient for a SAMS. CQL defined 3 types of sliding windows on streams, time-based windows, tuple-based windows, and partition-based window. A SAMS uses only time-based windows, which can be easily formulated as SQL predicates. In a ARGUS query plan, represented as a stored procedure, contains only operators and materialized intermediate results, but no queues. ARGUS intermediate results are stored in DBMS tables, and correspond to STREAM's synopses. To allow incremental evaluation, each ARGUS intermediate result storage comprises two parts, the main part that stores intermediate results for the historical data, and the delta part that stores the intermediate results for the new data. In STREAM, the synopses are always up to the current state, and the data changes are encoded as insertions and deletions to the queues. Bearing the similarities in the query languages, query plan structures, and constraint exploiting methods, the main differences between ARGUS and STREAM include: 1) ARGUS exploits full-fledged DBMS facilities, 2) ARGUS has no queues in query plans, 3) ARGUS uses SQL as the query language, 4) ARGUS can process historical data when continuous queries are registered after input streams have generated data tuples. STREAM doesn't not process historical data. 5) ARGUS is a prototype SAMS, and STREAM is a general-purpose DSMS.

Berkeley TelegraphCQ [14][46] [39] is a general-purpose DSMS that stresses adaptive query processing. The design is based on their prototyping Telegraph project, which builds an Adaptive Dataflow Architecture for supporting a wide variety of data-intensive, networked applications. The core concept of Telegraph's adaptive processing is the Eddy [5]. An Eddy performs scheduling of tuples by routing them through the operators that make up the query plan. The query plan is dynamically re-ordered to match current system conditions. This is accomplished by Eddies' tuple routing policies, with some overhead, that attempt to discover which operators are fast and selective, and those operators are scheduled first. Telegraph's components and development were described in various papers. These include Fjords [52], the inter-module communication API; Flux [72], the pluginable module to support distributed processing; SteM [65], the temporary repository of tuples to store state information during the plan execution; CACQ [53], the modification of Eddies to execute multiple queries simultaneously by having the Eddy execute a single super query corresponding to the disjunction of all the individual queries; and PSoup [15], the extension of CACQ by treating data and queries symmetrically to allow new queries to access historical data. The approach of TelegraphCQ is adapting the architecture of PostgreSQL, a DBMS code base, to enable shared processing of continuous queries over streaming sources, and then adding the continuous query processing functionalities which have been studied in

the Telegraph project (TelegraphCQ Executor and Wrapper, etc.) to the system. TelegraphCQ's query language is an extension of SQL with a for-loop construct to support window specifications. Because TelegraphCQ explores a very different approach to the query processing, the two systems, ARGUS and TelegraphCQ, are different in terms of many aspects including incremental query evaluation and computation sharing. TelegraphCQ's query plan is a set of operators and SteMs surrounding an Eddy which performs tuple routing. ARGUS query plan is a procedural push-based network. TelegraphCQ's computation sharing is achieved by allowing one Eddy to run a group of similar queries to share SteMs and filters. ARGUS plan sharing will be achieved by rewriting query plans to share the materialized intermediate results.

Aurora [1][12] is a general-purpose DSMS that stresses reasonable system response time. Users may define various Quality of Service (QoS) specifications for queries that describe the expectations of service quality degradation when the system is overloaded. QoS specifications guide Aurora to do appropriate operator scheduling, storage management, load shedding, and approximation, etc. Different to other systems, Aurora uses a procedural query language with a GUI interface to allow users to specify continuous queries as networks of connected operator boxes. Aurora dynamically optimizes the networks by identifying and optimizing their subnetworks. The main differences between ARGUS and Aurora include: 1) ARGUS exploits full-fledged DBMS facilities, 2) ARGUS uses the declarative query language, SQL, 3) ARGUS is a prototype SAMS.

OpenCQ [50], and WebCQ [51] at Georgia Tech, and NiagaraCQ [19] [73] at Wisconsin are continuous query systems for Internet databases. All these systems carry out incremental query evaluations over data changes. OpenCQ stresses the support for distributed computing with flexible coupling modes, and the support for INSERT, DELETE, and UPDATE operations to data sources instead of the append-only operations. WebCQ addresses the scalability issue over a large number of queries with Trigger Grouping [77], which is similar to predicate indexing [38]. NiagaraCQ features multiple query optimization by incrementally grouping and regrouping queries of similar structures. Grouped queries share the materialized intermediate results. ARGUS's computation sharing scheme is going to be implemented in the similar way. Different to above systems, 1) ARGUS is assumed to process homogeneous data tuples instead of XML files, 2) ARGUS supports more complex query semantics, particularly grouping, aggregation, and post-aggregation, 3) ARGUS is designed to process high-volume append-only data streams with rare matching. A lot of efforts have been put on to reducing resource requirement, and let the underlying DBMS to realize the opertors, scheduling, etc.

Alert and Tapestry are mentioned and compared with ARGUS in Section 3. Active database systems [34][85][13][54][67] allow users to specify, in the form of rules, actions to be performed upon changes of database states.

Gigascope [22] is a stream database system designed for network applications, such as traffic analysis and router configuration analysis. Special optimization techniques are applied, such as even pushing query operations into network interface cards. To turn block operators into non-blocking stream operators, Gigascope made some restrictions on joins and aggregations. Tribeca [75] is an earlier stream database system for network traffic analysis. Tribeca's executor is tuned for sequential I/O and the optimizer is focused toward memory and processor limitations rather than join ordering and access path selection.

Tukwila [43] is an adaptive query processing system for Internet applications. It incorporates event-condition-action rules to detect unexpected conditions, such as failed sources, and can return to the query optimizer to re-optimize the remainder of a query plan. Other work includes Hancock and Tangram. Hancock [21] computes salient features (signatures) from stream data efficiently. Tangram [45] was implemented in Prolog.

There is a large body of work related to continuous query processing over data streams. Query scrambling [80] reschedules the operations of a query during its execution on-the-fly. XJoin [79], Ripple Join [35] are non-blocking join operators developed for stream processing. Rate-Based query optimization [83] aims at maximizing the output rate of query plans based on the rate estimates of the streams in the query evaluation tree rather than the sizes of intermediate results. [49] provides a formalism on incremental query evaluation over data changes.

There are several database research directions related to streaming data processing. Materialized view maintenance [10][44][31][64][28] concerns the techniques of refreshing the views when base relations are changed. Sequence databases [70][69][71] model the logical ordering of sequence data, and applying the information to the optimization of sequence data queries. Time-series databases [25] are interested in the behaviors of subsequences in a stream of time-ordered data items. Temporal databases [74][24][86][87][11][29] stress on maintaining temporal versions of databases and their evolution. Index selection [2][16][81][36][33][17][84] targets automatic database tuning. Processing of queries on streaming XML data [32][59][3][61][23] mostly explores variant finite state automata (FSA) to find matches in XML files efficiently.

8 Conclusion and Future Work

For a Stream Anomaly Monitoring System that monitors thousands of queries when data continue to arrive rapidly, scalable system design and quickest responses are the key features to success. In this paper, we described our efforts on the ARGUS Profile System. The main approach is the combination of the Rete's incremental query evaluation techniques and the traditional DBMS. We implemented two improvements, namely, Transitivity Inference, and User-Defined Join Priorities. The preliminary results show that this approach is effective and efficient comparing to pure traditional DBMS approaches. Here, we outline several future work directions that we will explore.

8.1 Rete Network Optimization

Query optimization based on cost models has been well studied since the seminal paper [68]. [37] studied applying the cost model to Gator. Similar techniques can be applied to Rete network optimization. Partial Rete Generation in Section 6, and User-Defined Join Priorities in Section 5, are special cases of optimization. There are two main differences between the Rete optimization and the traditional query optimization. Rete optimization preserves the goal of optimizing the join ordering as in the traditional optimization, but sheds off the concern for choice of access methods (A Rete Optimizer doesn't need to consider the choice between merge join or nested-loop join, etc.). Instead, it faces the choice of materializing intermediate results or not. We will study appropriate cost models and implement the optimization module with the cost models.

8.2 Computation Sharing

It is observed that when thousands of queries are present in a system, many share common predicates. Sharing computation of the common predicates among these queries will significantly reduce the system response time [38], [19], [51]. As observed in [19], computation sharing is easier in a materialization-based processing model, such as Rete networks, than in a pipeline-based model. Predicate indexing [38] and sentinel grouping [51] are efficient ways for computation sharing. We will study similar techniques to allow computation sharing among Rete networks.

8.3 Incremental Aggregation

Aggregation operators can be classified as sort-based or non-sort-based. A sort-based aggregation operator, such as MEDIUM, PERCENTILE, and RANK, requires the whole group of data items to be sorted. A non-sort-based aggregation operator, such as SUM, COUNT, MIN, MAX, AVERAGE, and VARIANCE, can be incrementally evaluated by preserving sufficient statistics. For example, by preserving the SUM and COUNT, up-to-date AVERAGE can be calculated without accessing the historical data. We will implement incremental aggregate evaluation techniques into Rete networks for the non-sort-based operators.

8.4 Using Time Windows

To handle data streams of potentially unbounded sizes, we have to apply time windows to the data streams. In our system, the old data beyond a large threshold time window is discarded. However, a time window is not reluctantly accepted to overcome system limitations, but also provides desired query semantics [7]. Like in many stream processing applications, in a SAMS, recent data is more important and relevant, for example, a query monitoring endemic disease outbreaks. For all the join queries we have studied so far, there are explicit window specifications. We will study the techniques to identify the largest time window in a set of queries, so the old data beyond the time window can be discarded even it is within the system threshold.

8.5 Enhancing Transitivity Inference

We have shown that Transitivity Inference is an efficient improvement when it is applicable to highly selective predicates, which we assume is not rare in a SAMS. There are various directions to enhance the inference capabilities. Certain kinds of inference on function-present predicates require knowledge of function monotone properties. We will add such inference power by exploring monotone properties of commonly-used functions. We will also consider the kinds of inference involving multiple attributes across multiple tables.

8.6 Index Selection

As shown in Section 6, choice of indexing is a tradeoff balancing the benefits of query evaluation and the cost of maintenance. Guided by appropriate cost models, it is possible to automate the decision on whether indexes should be used or not and automate the optimal index selections. Inspired by research on automatic index selection [2][16][81][36][33][17][84], we will study and hope to develop a cost model that maximize the utility of indexes in the premise of a SAMS. This model is a function of three sets of parameters: indexes, a set of resident queries, and the data streams. As a prerequisite, we need to formalize the characteristics of the queries and the behaviors of data streams, such as the arrival rates of data, data statistics, and the effect of data truncation based on time windows.

Our work on ARGUS Profile System continues. By exploring more techniques including above mentioned ones, we hope that we will make ARGUS a more scalable and faster SAMS, and provide some insights and experience for stream data processing.

A A Sample Rete Network Procedure for Example 4

Following is a sample Rete procedure for Example 4

```
CREATE OR REPLACE PROCEDURE QUERY_1() AS
bP1 BOOLEAN;
bP2 BOOLEAN;
bP3 BOOLEAN;
bP4 BOOLEAN;
bP5 BOOLEAN;
bP4_1 BOOLEAN;
bP4_2 BOOLEAN;
bP4_3 BOOLEAN;
bP5_1 BOOLEAN;
bP5_2 BOOLEAN;
bP5_3 BOOLEAN;

BEGIN

bP1 := FALSE;
bP2 := FALSE;
bP3 := FALSE;
bP4 := FALSE;
bP5 := FALSE;
bP4_1 := FALSE;
bP4_2 := FALSE;
bP4_3 := FALSE;
bP5_1 := FALSE;
bP5_2 := FALSE;
bP5_3 := FALSE;
```

INSERT INTO Q1_V1
SELECT tranid, rbank_aba, benef_account, amount, tran_date
FROM TEMP_TRANSACTION t1;
WHERE $t1.type\_code = 1000$ AND
$t1.amount > 1000000$;
bP1 := SQL%FOUND;

INSERT INTO Q1_V2
SELECT tranid, sbank_aba, orig_account,
amount, tran_date, rbank_aba, benef_account
FROM TEMP_TRANSACTION t2;
WHERE $t2.type\_code = 1000$;
bP2 := SQL%FOUND;

INSERT INTO Q1_V3
SELECT tranid, sbank_aba, orig_account, amount, tran_date
FROM TEMP_TRANSACTION t3;
WHERE $t3.type\_code = 1000$;
bP3 := SQL%FOUND;

IF (bP2) THEN
INSERT INTO Q1_V4
SELECT v1.tranid tranid_1, v2.tranid tranid_2,
v2.rbank_aba, v2.benef_account, v2.amount, v2.tran_date
FROM Q1_V1 v1, temp_Q1_V2 v2
WHERE $v1.rbank\_aba = v2.sbank\_aba$ AND
$v1.benef\_account = v2.orig\_account$ AND
$v1.amount * 0.5 < v2.amount$ AND
$v1.tran\_date <= v2.tran\_date$ AND
$v1.tran\_date + 10 >= v2.tran\_date$;
bP4_1 := SQL%FOUND;
END IF;

IF (bP1) THEN
INSERT INTO Q1_V4
SELECT v1.tranid tranid_1, v2.tranid tranid_2,
v2.rbank_aba, v2.benef_account, v2.amount, v2.tran_date
FROM temp_Q1_V1 v1, Q1_V2 v2
WHERE $v1.rbank\_aba = v2.sbank\_aba$ AND
$v1.benef\_account = v2.orig\_account$ AND
$v1.amount * 0.5 < v2.amount$ AND
$v1.tran\_date <= v2.tran\_date$ AND
$v1.tran\_date + 10 >= v2.tran\_date$;

bP4_2 := SQL%FOUND;
END IF;

IF (bP1 AND bP2) THEN
INSERT INTO Q1_V4
SELECT v1.tranid tranid_1, v2.tranid tranid_2,
v2.rbank_aba, v2.benef_account, v2.amount, v2.tran_date
FROM temp_Q1_V1 v1, temp_Q1_V2 v2
WHERE $v1.rbank\_aba = v2.sbank\_aba$ AND
$v1.benef\_account = v2.orig\_account$ AND
$v1.amount * 0.5 < v2.amount$ AND
$v1.tran\_date <= v2.tran\_date$ AND
$v1.tran\_date + 10 >= v2.tran\_date$;
bP4_3 := SQL%FOUND;
END IF;

bP4 := bP4_1 OR bP4_2 OR bP4_3;

IF (bP4) THEN
INSERT INTO Q1_V5
SELECT v4.tranid_1, v4.tranid_2, v3.tranid tranid_3
FROM Q1_V3 v3, temp_Q1_V4 v4
WHERE $v3.sbank\_aba = v4.rbank\_aba$ AND
$v3.orig\_account = v4.benef\_account$ AND
$v3.amount = v4.amount$ AND
$v3.tran\_date >= v4.tran\_date$ AND
$v3.tran\_date <= v4.tran\_date + 10$;
bP5_1 := SQL%FOUND;
END IF;

IF (bP3) THEN
INSERT INTO Q1_V5
SELECT v4.tranid_1, v4.tranid_2, v3.tranid tranid_3
FROM temp_Q1_V3 v3, Q1_V4 v4
WHERE $v3.sbank\_aba = v4.rbank\_aba$ AND
$v3.orig\_account = v4.benef\_account$ AND
$v3.amount = v4.amount$ AND
$v3.tran\_date >= v4.tran\_date$ AND
$v3.tran\_date <= v4.tran\_date + 10$;
bP5_2 := SQL%FOUND;
END IF;

IF (bP3 AND bP4) THEN

```
 INSERT INTO Q1_V5
 SELECT v4.tranid_1, v4.tranid_2, v3.tranid tranid_3
 FROM temp_Q1_V3 v3, temp_Q1_V4 v4
 WHERE v3.sbank_aba = v4.rbank_aba AND
 v3.orig_account = v4.benef_account AND
 v3.amount = v4.amount AND
 v3.tran_date >= v4.tran_date AND
 v3.tran_date <= v4.tran_date + 10;
 bP5_3 := SQL%FOUND;
END IF;

bP5 := bP5_1 OR bP5_2 OR bP5_3;

IF (bP5) THEN
 DBMS_OUTPUT.PUT_LINE('Alert!!!');
END IF;

END;
```

B Sample Rete Network DDLs for Example 4

Following DDLs are for the Rete network of Example 4 shown in Appendix A.

CREATE TABLE Q1_V1 AS
SELECT tranid, rbank_aba, benef_account, amount, tran_date
FROM tran1.transaction
WHERE $type\_code = 1000$ AND
$amount > 1000000$;

CREATE TABLE Q1_V2 AS
SELECT tranid, sbank_aba, orig_account, amount,
tran_date, rbank_aba, benef_account
FROM tran1.transaction
WHERE $type\_code = 1000$;

CREATE TABLE Q1_V3 AS
SELECT tranid, sbank_aba, orig_account, amount, tran_date
FROM tran1.transaction
WHERE $type\_code = 1000$;

CREATE TABLE Q1_V4 AS
SELECT v1.tranid tranid_1, v2.tranid tranid_2,
v2.rbank_aba, v2.benef_account, v2.amount, v2.tran_date
FROM Q1_V1 v1, Q1_V2 v2
WHERE $v1.rbank\_aba = v2.sbank\_aba$ AND
$v1.benef\_account = v2.orig\_account$ AND
$v1.amount * 0.5 < v2.amount$ AND
$v1.tran\_date <= v2.tran\_date$ AND
$v1.tran\_date + 10 >= v2.tran\_date$;

CREATE TABLE Q1_V5 AS
SELECT v4.tranid_1, v4.tranid_2, v3.tranid tranid_3
FROM Q1_V3 v3, Q1_V4 v4
WHERE $v3.sbank\_aba = v4.rbank\_aba$ AND
$v3.orig\_account = v4.benef\_account$ AND
$v3.amount = v4.amount$ AND
$v3.tran\_date >= v4.tran\_date$ AND
$v3.tran\_date <= v4.tran\_date + 10$;

C A Sample Rete Network Procedure for Example 5

Following is the sample Rete network procedure for Example 5

```
CREATE OR REPLACE PROCEDURE QUERY_3 AS
 bP2 BOOLEAN;

BEGIN
 bP2 := FALSE;

 INSERT INTO Q3_V2
 SELECT r.rbank_aba rbank_aba, s.sbank_aba sbank_aba,
r.tran_date tran_date, r.rsum rsum, s.ssum ssum
FROM rbank_money_q3 r, sbank_money_q3 s
WHERE ((r.rbank_aba = s.sbank_aba) AND
(r.tran_date = s.tran_date));

 INSERT INTO temp_Q3_V1
 SELECT *
 FROM Q3_V2
 MINUS
 SELECT *
 FROM Q3_V1;
 bP2 := SQL%FOUND;
 IF (bP2) THEN
 DELETE Q3_V1;
 INSERT INTO Q3_V1
 SELECT *
 FROM Q3_V2;
 DELETE Q3_V2;
 END IF;

IF (bP2) THEN
 DBMS_OUTPUT.PUT_LINE('Alert!!!');
END IF;

END;
```

D A Sample Rete Network Procedure for Example 3

Following is the sample Rete network procedure for Example 3

```
CREATE OR REPLACE PROCEDURE QUERY_4 AS
 bP1 BOOLEAN;
 bP2 BOOLEAN;
 bP2_1 BOOLEAN;
 bP2_2 BOOLEAN;
 bP2_3 BOOLEAN;
 bP3 BOOLEAN;
 bP4 BOOLEAN;
 var1 INTEGER;
BEGIN
```

```
 bP1 := FALSE;
 bP2 := FALSE;
 bP2_1 := FALSE;
 bP2_2 := FALSE;
 bP2_3 := FALSE;
 bP3 := FALSE;
 bP4 := FALSE;

SELECT COUNT(*) INTO var1
FROM TEMP_TRANSACTION;
IF (var1 > 0)
THEN
 bP1 := TRUE;
END IF;

IF (bP1) THEN
 INSERT INTO temp_Q4_V2
 SELECT r1.amount u_r__amount__4, r1.rbank_aba u_r__rbank_aba__2,
 r1.tran_date + 10 u_r__tran_date__10__12, r1.tranid u_r__tranid__1,
 r1.benef_account u_r__benef_account__3, r1.tran_date u_r__tran_date__10
 FROM TEMP_TRANSACTION r1
 WHERE r1.amount > 1000000;
 bP3 := SQL%FOUND;
END IF;

IF (bP1) THEN
 INSERT INTO temp_Q4_V1
 SELECT r1.sbank_aba u_s__sbank_aba__8, r1.amount u_s__amount__6,
 r1.orig_account u_s__orig_account__9, r1.tran_date u_s__tran_date__11,
 r2.u_r__amount__4 u_r__amount__4, r2.u_r__rbank_aba__2 u_r__rbank_aba__2,
 r2.u_r__tran_date__10 + 10 u_r__tran_date__10__12,
 r2.u_r__tranid__1 u_r__tranid__1, r2.u_r__benef_account__3 u_r__benef_account__3,
 r2.u_r__tran_date__10 u_r__tran_date__10
 FROM TEMP_TRANSACTION r1, Q4_V2 r2
 WHERE (r2.u_r__rbank_aba__2 = r1.sbank_aba AND
 r2.u_r__benef_account__3 = r1.orig_account AND
 r2.u_r__tran_date__10 <= r1.tran_date AND
 r1.tran_date <= r2.u_r__tran_date__10 + 10);
 bP2_1 := SQL%FOUND;
END IF;

IF (bP3) THEN
 INSERT INTO temp_Q4_V1
```

```
 SELECT r1.sbank_aba u_s__sbank_aba__8, r1.amount u_s__amount__6,
 r1.orig_account u_s__orig_account__9, r1.tran_date u_s__tran_date__11,
 r2.u_r__amount__4 u_r__amount__4, r2.u_r__rbank_aba__2 u_r__rbank_aba__2,
 r2.u_r__tran_date__10 + 10 u_r__tran_date__10__12,
 r2.u_r__tranid__1 u_r__tranid__1, r2.u_r__benef_account__3 u_r__benef_account__3,
 r2.u_r__tran_date__10 u_r__tran_date__10
 FROM temp_Q4_V2 r2, transaction_copy r1
 WHERE (r2.u_r__rbank_aba__2 = r1.sbank_aba AND
 r2.u_r__benef_account__3 = r1.orig_account AND
 r2.u_r__tran_date__10 <= r1.tran_date AND
 r1.tran_date <= r2.u_r__tran_date__10 + 10);
 bP2_2 := SQL%FOUND;
END IF;

IF (bP1 AND bP3) THEN
 INSERT INTO temp_Q4_V1
 SELECT r1.sbank_aba u_s__sbank_aba__8, r1.amount u_s__amount__6,
 r1.orig_account u_s__orig_account__9, r1.tran_date u_s__tran_date__11,
 r2.u_r__amount__4 u_r__amount__4, r2.u_r__rbank_aba__2 u_r__rbank_aba__2,
 r2.u_r__tran_date__10 + 10 u_r__tran_date__10__12,
 r2.u_r__tranid__1 u_r__tranid__1, r2.u_r__benef_account__3 u_r__benef_account__3,
 r2.u_r__tran_date__10 u_r__tran_date__10
 FROM TEMP_TRANSACTION r1, temp_Q4_V2 r2
 WHERE (r2.u_r__rbank_aba__2 = r1.sbank_aba AND
 r2.u_r__benef_account__3 = r1.orig_account AND
 r2.u_r__tran_date__10 <= r1.tran_date AND
 r1.tran_date <= r2.u_r__tran_date__10 + 10);
 bP2_3 := SQL%FOUND;
END IF;

 bP2 := bP2_1 OR bP2_2 OR bP2_3;

IF (bP3) THEN
 INSERT INTO Q4_V2
 SELECT *
 FROM temp_Q4_V2;
END IF;

IF (bP2) THEN
 INSERT INTO Q4_V1
 SELECT *
 FROM temp_Q4_V1;
END IF;
```

```
IF (bP2) THEN
 INSERT INTO Q4_V4
 SELECT u_r__tranid__1 tranid, u_r__rbank_aba__2 rbank_aba,
 u_r__benef_account__3 benef_account, SUM(u_r__amount__4) ramount,
 SUM(u_s__amount__6) samount
FROM Q4_V1 r
GROUP BY u_r__tranid__1, u_r__benef_account__3, u_r__rbank_aba__2
HAVING SUM(u_s__amount__6) > SUM(u_r__amount__4) * 0.5;

 INSERT INTO temp_Q4_V3
 SELECT *
 FROM Q4_V4
 MINUS
 SELECT *
 FROM Q4_V3;
 bP4 := SQL%FOUND;

 IF (bP4) THEN
 DELETE Q4_V3;
 INSERT INTO Q4_V3
 SELECT *
 FROM Q4_V4;
 DELETE Q4_V4;
 END IF;
END IF;

IF (bP4) THEN
 DBMS_OUTPUT.PUT_LINE('Alert!!!');
END IF;

DELETE temp_Q4_V1;
DELETE temp_Q4_V2;
DELETE temp_Q4_V3;

END;
```

References

[1] Daniel J. Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August, 2003.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *Proceedings of 26th International Conference on Very Large Data Bases*, Cairo, Egypt, 2000.

[3] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 53–64, Cairo, Egypt, September, 2000.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report 2003-68, Stanford University, October, 2003.

[5] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*, pages 261–272, Dallas, Texas, May, 2000.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 253–264, San Diego, California, June, 2003.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, Madison, Wisconsin, June, 2002.

[8] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *ACM SIGMOD Record*, 30(3):109–120, September, 2001.

[9] Ruth Baylis and Kathy Rich. *Oracle9i Database Administrator's Guide Release 2 (9.2)*. Oracle Corporation, Redwood Shores, CA, 2002.

[10] Jose A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Database Systems (TODS)*, 14(3):369–400, September, 1989.

[11] Michael H. Bohlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4):53–60, December, 1995.

[12] Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China, August, 2002.

[13] Sharma Chakravarthy. Architectures and Monitoring Techniques for Active Databases: An Evaluation. Technical Report TR-92-041, CISE Dept., University of Florida, 1992.

[14] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR)*, January, 2003.

[15] Sirish Chandrasekaran and Michael J. Franklin. Streaming Queries over Streaming Data. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 203–214, Hong Kong, China, August, 2002.

[16] Surajit Chaudhuri and Vivek Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 146–155, Athens, Greece, 1997.

[17] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 1–10, Cairo, Egypt, 2000.

[18] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of the 18th International Conference on Data Engineering*, pages 345–356, San Jose, California, February, 2002.

[19] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*, pages 379–390, Dallas, Texas, May, 2000.

[20] Robert Corbett, Rick Ohnemus, and Jake Donham. Perl-byacc. In *http://mirrors.valueclick.com/pub/perl/CPAN/src/misc/*, 1998.

[21] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, Boston, Massachusetts, 2000.

[22] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 647–651, San Diego, California, June, 2003.

[23] Yanlei Diao and Michael J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26(1):41–48, March, 2003.

[24] Richard T. Snodgrass et al. TSQL2 Language Specification. *SIGMOD Record*, 23(1):65–86, August, 1994.

[25] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast Subsequence Matching in Time-Series Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of data*, pages 419–429, Minneapolis, Minnesota, May, 1994.

[26] Steven Feuerstein. *Oracle PL/SQL Programming*. O'Reilly & Associates, Inc, 1995.

[27] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, Septempber, 1982.

[28] Hector Garcia-Molina, Wilburt Juan Labio, and Jun Yang. Expiring Data in a Warehouse. In *Proceedings of 24th International Conference on Very Large Data Bases*, pages 500–511, New York City, New York, August, 1998.

[29] Iqbal A. Goralwalla, M. Tamer Ozsu, and Duane Szafron. An object-oriented framework for temporal data models. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*, pages 1–35. Springer Verlag, 1998.

[30] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[31] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of 5th International Conference on Extending Database Technology*, pages 140–144, Avignon, France, March, 1996.

[32] Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 419–430, San Diego, California, June, 2003.

[33] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index Selection for OLAP. In *Proceedings of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[34] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Startburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March, 1990.

[35] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of data*, pages 287–298, Philadelphia, Pennsylvania, June, 1999.

[36] Michael Hammer and Arvola Chan. Index Selection in a Self-Adaptive Data Base Management System. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of data*, pages 1–8, Washington, D.C., 1976.

[37] Eric N. Hanson, Sreenath Bodagala, and Ullas Chadaga. Optimized Trigger Condition Testing in Ariel Using Gator Networks. Technical Report TR-97-021, CISE Dept., University of Florida, November, 1997.

[38] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable Trigger Processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, Sydney, Austrialia, March, 1999.

[39] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive Query Processing: Technology in Evolution. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2):7–18, June, 2000.

[40] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pages 268–277, Denver, Colorado, May, 1991.

[41] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of data*, pages 312–321, Atlantic City, NJ, May, 1990.

[42] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of data*, pages 299–310, Philadelphia, Pennsylvania, June, 1999.

[43] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive Query Processing for Internet Applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2):19–26, June, 2000.

[44] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View Maintenance Issues for the Chronicle Data Model (Extended Abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 113–124, San Jose, California, May, 1995.

[45] Douglas Stott Parker Jr., Richard R. Muntz, and H. Lewis Chau. The Tangram Stream Query Processing System. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 556–563, Los Angeles, California, February, 1989.

[46] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: An Architectural Status Report. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26(1):11–18, March, 2003.

[47] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of 12th International Conference on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August, 1986.

[48] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc, 1995.

[49] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential Evaluation of Continual Queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, China, May, 1996.

[50] Ling Liu, Calton Pu, and Wei Tang. Continual Queires for Internet Scale Event-Driven Information Delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):583–590, August, 1999.

[51] Ling Liu, Wei Tang, David Buttler, and Calton Pu. Information Monitoring on the Web: A Scalable Solution. *World Wide Web Journal*, 5(4), 2002.

[52] Samuel Madden and Michael J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555–566, San Jose, California, February, 2002.

[53] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 49–60, Madison, Wisconsin, June, 2002.

[54] Dennis R. McCarthy and Umeshwar Dayal. The Architecture of an Active Data Base Management System. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, 1989.

[55] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for IA Production Systems*. Morgan Kaufmann, 1990.

[56] Daniel P. Miranker. Treat: A better match algorithm for ai production systems. Technical Report AI TR87-58, The University of Texas at Austin, Austin, TX, July, 1987.

[57] Daniel P. Miranker and David A. Brant. An algorithmic basis for integrating production systems and large databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 353–360, Los Angeles, California, February, 1990.

[58] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, , and Rohit Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR)*, pages 245–256, January, 2003.

[59] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML Data on the Web. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of data*, pages 437–448, Santa Barbara, California, May, 2001.

[60] Parick O'Neil and Dallan Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of data*, pages 38–49, Tucson, Arizona, June, 1997.

[61] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data . In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 431–442, San Diego, California, June, 2003.

[62] Mark W. Perlin. The match box algorithm for parallel production system match. Technical Report CMU-CS-89-163, Carnegie Mellon University, Pittsburgh, PA, May, 1989.

[63] Viswanath Poosala, Vannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of data*, pages 294–305, Montreal, Canada, June, 1996.

[64] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of 4th International Conference on Parallel and Distributed Information Systems*, pages 158–169, December, 1996.

[65] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proceedings of the 19th International Conference on Data Engineering*, pages 353–, Bangalore, India, March, 2003.

[66] John Russell, T. Brooksfuller, T. Burroughs, M. Cowan, J. Levinger, R. Moran, and R. Strohm. *Oracle9i Application Developer's Guide - Fundamentals, Release 2 (9.2)*. Oracle Corporation, Redwood Shores, CA, 2002.

[67] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 469–478, Barcelona, Spain, September, 1991.

[68] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of data*, pages 23–34, Boston, MA, May, 1979.

[69] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the 11th International Conference on Data Engineering*, pages 232–239, Taipei, Taiwan, March, 1995.

[70] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of data*, pages 430–441, Minneapolis, Minnesota, May, 1994.

[71] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 99–110, Bombay, India, September, 1996.

[72] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the 19th International Conference on Data Engineering*, pages 25–36, Bangalore, India, March, 2003.

[73] Jayavel Shanmugasundaram, Kristin Tufte, David DeWitt, Jeffrey Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *Proceedings of the 2000 Intl. Workshop on the Web and Databases*, pages 17–22, May, 2000.

[74] Richard Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of data*, pages 236–246, Austin, Texas, May, 1985.

[75] Mark Sullivan and Andrew Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proceedings of the USENIX Annual Technical Conference*, pages 13–24, New Orleans, LA, June, 1998.

[76] Mark Sullivan and Andrew Heybey. Tribeca: A Stream Database Manager for Network Traffic Analysis. In *Proceedings of 22nd International Conference on Very Large Data Bases*, page 594, Bombay, India, September, 1996.

[77] Wei Tang, Ling Liu, and Calton Pu. Trigger Grouping: A Scalable Approach to Large Scale Information Monitoring. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications (NCA-03)*, Cambridge, MA, April, 2003.

[78] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of data*, pages 321–330, San Diego, California, June, 1992.

[79] Tolga Urhan and Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 23(2):27–33, June, 2000.

[80] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based Query Scrambling for Initial Delays. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of data*, pages 130–141, Seattle, Washington, June, 1998.

[81] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering*, pages 101–110, San Diego, California, 2000.

[82] Philippe Verdret. Perl Lexer. In *http://search.cpan.org/author/PVERD/ParseLex-2.15/*, 1999.

[83] Stratis D. Viglas and Jeffrey F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 37–48, Madison, Wisconsin, June, 2002.

[84] Gerhard Weikum, Axel Moenkeberg, Chrstof Hasse, and Peter Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

[85] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems*. Morgan Kaufmann, 1996.

[86] Jun Yang and Jennifer Widom. Temporal View Self-Maintenance. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 395–412, Konstanz, Germany, March, 2000.

[87] Jun Yang and Jennifer Widom. Incremental Computation and Maintenance of Temporal Aggregates. *International Journal on Very Large Data Bases (VLDB Journal)*, 12(3):262–283, October, 2003.