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

Chun Jin[1], Jaime Carbonell[1], and Phil Hayes[2]

[1] Language Technologies Institute, School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213 USA
{cjin, jgc}@cs.cmu.edu
[2] Dynamix Technologies, 12330 Perry Highway, Wexford, PA 15090 USA
phayes@dynamixtechnologies.com

Abstract. Efficient processing of complex streaming data presents multiple challenges, especially when combined with intelligent detection of hidden anomalies in real time. We label such systems Stream Anomaly Monitoring Systems (SAMS), and describe the CMU/Dynamix ARGUS system as a new kind of SAMS to detect rare but high value patterns combining streaming and historical data. Such patterns may correspond to hidden precursors of terrorist activity, or early indicators of the onset of a dangerous disease, such as a SARS outbreak. Our method starts from an extension of the RETE algorithm for matching streaming data against multiple complex persistent queries, and proceeds beyond to transitivity inferences, conditional intermediate result materialization, and other such techniques to obtain both accuracy and efficiency, as demonstrated by the evaluation results outperforming classical techniques such as a modern DMBS.

1 Introduction

Efficient processing of complex streaming data presents multiple challenges. Among data intensive stream applications, we identify an important sub-class which we call Stream Anomaly Monitoring Systems (SAMS). A SAMS monitors transaction data streams or other structured data streams and alerts when anomalies or potential hazards are detected. The system is expected to deal with very-high data-rate streams, many millions of records per day, yet the matches of the anomaly conditions should be very infrequent. However, the matches may generate very high-urgency alerts, may be fed to decision-making systems, and may invoke significant actions. The conditions for anomalies or potential hazards are formulated as persistent queries over data streams by experienced analysts. The data streams are composed of homogeneous records such as money transfer transaction records or hospital inpatient admission records.

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 between big money transfer transactions connected to suspected people or organizations. 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 between trading transactions with certain features may draw an analyst's attention to check whether insider information is being illegally used.

In this paper, we are concerned with optimal incremental evaluation plans of rarely matching persistent queries over high data rate streams and large-volume historical data archives. We focus on exploring the very-high-selectivity query property to produce good/optimal incremental evaluation plans to solve the performance problem posed by the very-large-volume historical data and very-high stream data rates.

The basic algorithm for the incremental evaluation is a variant of the Rete algorithm which is widely used in rule-based production systems. The Rete match algorithm [7] is an efficient method for matching a large collection of patterns to a large collection of objects. By storing partially instantiated (matched) patterns, Rete saves a significant computation that would otherwise have to be re-computed repetitively in recursive matching of the newly produced working elements. The adapted Rete for stream processing adopts the same idea of storing intermediate results (partial results) of the persistent queries matched against the historical data. New data items arriving at the system may match with the intermediate results to significantly speed up producing the new query results. This is particularly useful when intermediate result size is much smaller than the size of the original data to be processed. This is exactly the case for many SAMS queries. The historical data volume is very large, yet intermediate results may be minimized by exploiting the very-high-selectivity query property.

ARGUS is a prototype SAMS that exploits the adapted Rete algorithm, and is built upon the platform of Oracle DBMS. It also explores transitivity inferences, and conditional intermediate result materialization, and will further incorporate complex computation sharing functionality and the Dynamix Matcher [6] into the system. The Dynamix Matcher is an integrated part of the ARGUS project that can perform fast simple-query filtering before joins and aggregations are processed by Rete; it has also been used for commercial applications. In ARGUS, a persistent query is translated into a procedural network of operators on streams and relations. Derived (intermediate) streams or relations are conditionally materialized as DBMS tables. 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.

In this paper, we present some SAMS query examples, describe the Rete-based ARGUS design and extensions, and conclude with preliminary performance results, and related work.

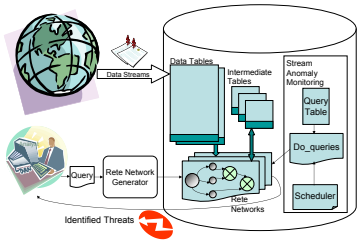

Fig. 1. ARGUS System Architecture

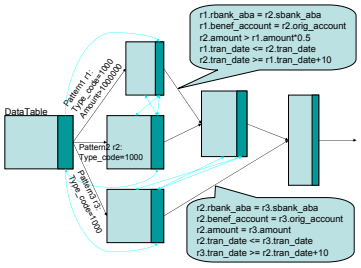

Fig. 2. A Rete network for Example 4

2 SAMS Query Examples

We choose the FedWire money transfer domain for illustration and experiments in this paper. It has a single data stream that contains money transfer transaction records. We present seven persistent query examples (one is also presented with the formulated

SQL query) which cover the SAMS query scope: selection, join, aggregation, and using views. For more details, and more thorough analysis of the results please see [11].

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

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

Example 3. For every big transaction, the analyst wants to check if the money stayed in the bank or left it within ten days by transferring out in several smaller transactions. 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.

Example 4. 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 using an intermediate bank.

SELECT $*$
FROM $transaction\, r1,\ transaction\, r2,$
$transaction\, r3$
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
(continue)
$r1.benef\_account = r2.orig\_account$ 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
$r3.tran\_date <= r2.tran\_date + 10;$

Example 5. Check whether any bank has incoming transactions of $100,000,000 or more and outgoing transactions of $50,000,000 or more on one particular day.

Example 6. Get the transactions of Citibank and Fleet on a particular day.

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

3 ARGUS Profile System Design

Figure 1 shows the SAMS dataflow and the ARGUS system architecture. Analysts selectively formulate the conditions of anomalies or potential hazards as persistent queries, and register them with the system. Data records in streams arrive continuously. Registered queries are scheduled periodic executions over the new data records, and return any new results as alerts. The ARGUS system contains two components, the database created on the Oracle DBMS, and the Rete construction module, ReteGenerator, which translates persistent queries into Rete networks. Wrapped in a stored procedure, a Rete network encodes an instance of the adapted Rete algorithm. Registering a persistent query in the database includes creating and initializing the intermediate tables based on the historical data, and storing and compiling the Rete network procedure.

QueryTable is a system table that records query information, one entry per query. Each 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. $Do\_queries()$ is a system level procedure that finds all the active Rete networks from QueryTable in the order of their priorities, and executes them one by one. New data arrive continuously and are appended to data tables. The active Rete networks are scheduled periodical runs on new data arrivals, and generate alerts when any persistent query matches the new data.

3.1 Adapted Rete Algorithm

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 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 with $O(n + m)$.

Figure 2 shows a Rete network for Example 4. A satisfied result set contains joins of three tuples each of which satisfies a set of selection predicates, identified as Pattern 1, Pattern 2, and Pattern 3, respectively. To allow incremental evaluation, each intermediate result storage comprises two parts, the main part that stores intermediate results for historical data, and the delta part that stores the intermediate results for new data.

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, a potential problem is that when any materialized intermediate table grows very large, thus requiring many I/O operations, the performance degrades severely. Fortunately, since queries are expected to be satisfied infrequently, there are usually highly selective conditions that make the intermediate tables fairly small. We investigated several optimization techniques to minimizing the sizes of intermediate result tables.

3.2 Translating SQL Queries into Rete Networks

A 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. For more details, please see [11].

ReteGenerator ReteGenerator contains three components: the SQL Parser, the Rete Topology Constructor, and the Rete Coder. The SQL Parser parses a query (a set of SQLs) to a set of parse trees. The Rete Topology Constructor rearranges the connections of nodes in the sub-parse tree of each SQL term to obtain the desired sub-Rete network topologies. And the Rete Coder generates the Rete network code and corresponding DDL statements in Oracle PL/SQL language by traversing the reconstructed parse trees and instantiating the code templates.

The Rete Topology Constructor takes three steps to construct the sub-Rete network topology for each SQL term based on its $where\_clause$ sub-parse tree. First, predicates are classified based on the tables they use. Second, the classified predicate sets are sorted based on the number of tables that the sets contain. Finally, the new subtree is reconstructed bottom-up. Single-table predicate sets correspond to leaf nodes. A new node joining two existing nodes is selectively created if a join predicate set exists. The process continues until all nodes are merged into a single node. Figure 3 shows the reconstructed $where\_clause$ subtree of the query Example 4.

Aggregation and Union An SQL term 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 the groupby/having clauses. Then the system does the operations of grouping/having on the whole input table, and finds the difference between the current results and the previous results. These grouping operations on whole input sets will be replaced by incremental aggregation in future.

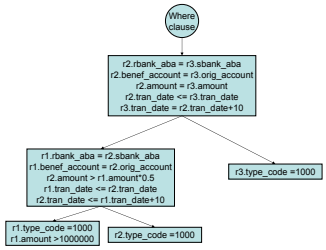

Fig. 3. The Reconstructed Where_clause Subtree for Example 4

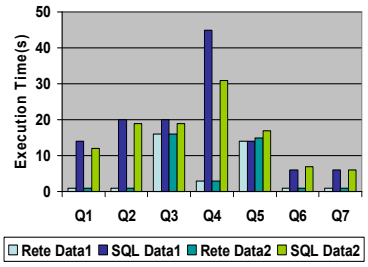

Fig. 4. Execution Times of Q1-Q7

3.3 Improvements on Rete Network

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 example, in Example 4, the query has the following conditions (the first is very selective): $r1.amount > 1000000$, $r2.amount > r1.amount{*}0.5$, and $r3.amount = r2.amount$. The first two conditions imply a selective condition on $r2$: $r2.amount > 500000$. Further, the third condition and the newly derived condition imply another selective condition on $r3$: $r3.amount > 500000$. These inferred conditions have significant impact on performance. The first level intermediate tables, filtered by the highly selective selection predicates, are made very small, which saves significant computation on subsequent joins.

Conditional Materialization If intermediate results grow large, for instance in join queries where single-table selection predicates are not selective and transitivity inference is not applicable, pipelined operation is preferable to materialization. Assume Transitivity Inference is not applicable by turning the module off, Example 4 is such a query. The two single-table selection predicates ($r2.type\_code = 1000$, $r3.type\_code = 1000$) are highly non-selective, the sizes of the intermediate results are close to that of the original data table. Aware of the table statistics, or indicated by users, such a materialization can be conditionally skipped, which we call Conditional Materialization. In our experiments, Rete network Q11 is a Conditional Rete network for Example 4 while

Q10 is another one for the same example. They are similar except that Q11 does not materialize the results of the two non-selective selection predicates.

User-Defined Join Priority Join priority specifies the join order that the Rete network should take. It is similar to the reordering of join operators in traditional query optimization. The ReteGenerator currently accepts user-defined join priority. We are working on applying query optimization techniques based on table statistics and cost models to automatically decide the optimal join order.

3.4 Current Research

We are designing complex extensions to incorporate computation sharing, cost-based optimization, incremental aggregation, and the Dynamix Matcher. Computation sharing among multiple queries adds much more complexity to the system. We are developing the schemes to index query predicates and predicate sets, and algorithms to identify and rearrange predicate sets to minimize intermediate result sizes in the shared networks. Cost-based optimization automatically decides the join order and the choice of conditional materialization based on table statistics. Incremental aggregation aggregates data items by maintaining sufficient statistics instead of the whole group items. For example, by preserving the SUM and COUNT, up-to-date AVERAGE can be calculated without accessing the historical data. The Dynamix Matcher is a fast query matching system for large-scale simple queries that exploits special data structures. We are working on the scheme of rerouting the partial query evaluations to Dynamix Matcher when they can be efficiently carried out by the Dynamix Matcher.

4 Experimental Results

4.1 Experiment Setting

The experiments compared performance of the prototype ARGUS to the Oracle DBMS that ARGUS was built upon, and were conducted on an HP computer with 1.7G CPU and 512M RAM, running Windows XP.

Data Conditions The data conditions are derived from a database of synthesized FedWire money transfer transactions. The database D contains 320,006 records. The timestamps of the data span 3 days. We split the data in two ways, and most of the experiments were conducted on both data conditions:

- Data1. Old data: the first $300,000$ records of D. New data: the remaining $20,006$ records of D. This data set provides alerts for most of the queries being tested.
- Data2. Old data: the first $300,000$ records of D. New data: the next $20,000$ records of D. This data set does not generate alerts for most of the queries being tested.

Queries We tested eleven Rete networks created for the seven queries described in Section 2. Q1-Q7 are the Rete networks for the seven examples created in a common setting, respectively: no hidden condition is added to the original queries, and Transitivity Inference module is turned on. Q8 and Q10 are variants of Example 2 and 4 that are generated without Transitivity Inference. Q9 is the variant of Example 4 whose original SQL query is enhanced with hidden conditions. Q11 is a Conditional Rete network of Example 4.

When running the original SQL queries, we combined the historical (old) data and the new data (stream). It takes some time for Rete networks to initialize intermediate results, yet it is a one-time operation. Rete networks provide incremental new results, while original SQL queries only provide whole sets of results.

4.2 Results Interpretation

Figure 4 summarizes the results of running the queries Q1-Q7 on the two data conditions. For most of the queries, Rete networks with Transitivity Inference gain significant improvements over directly running the SQL queries. For more details, please see [11].

Aggregation Q3 and Q5 are the two queries involving aggregations. Q3's Rete network has to join with the large original table. And Rete's incremental evaluation scheme is not applicable to Q5. This leads to limited or zero improvement of the Rete procedures. We expect incremental aggregation will provide noticeable improvements to these queries.

Transitivity Inference Example 2 and 4 are queries that benefit from Transitivity Inference. Figure 5 shows the execution times for these two examples. 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.

Note that in Figure 5 $SQL\ TI\ (Q9)$, Example 4 with manually added conditions, runs significantly faster than the original one. This suggests that this type of complex transitivity inference is not applied in the DBMS query optimization, and may be a potential new query optimization method for a traditional DBMS.

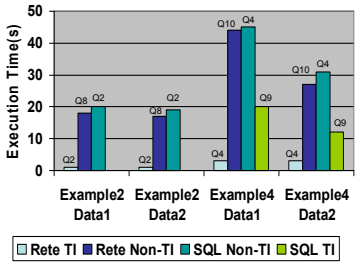

Fig. 5. "Rete TI": Rete generated with Transitivity Inference. "Rete Non-TI": Rete without Transitivity Inference. "SQL Non-TI": Original SQL query. "SQL TI": Original SQL query with hidden conditions manually added.

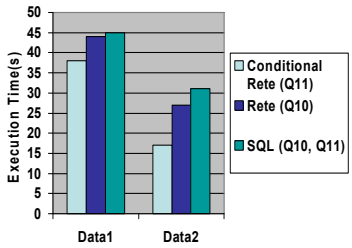

Fig. 6. Effect of Conditional Materialization. Comparing the execution times of Conditional Rete, Non-Conditional Rete, and SQL for Example 4 on Data1 and Data2.

Conditional Materialization Assume Transitivity Inference is not applicable by turning the module off for Example 4, we obtain a Rete network Q10, and a Conditional Rete network Q11. Figure 6 compares the execution times of the Conditional Rete network, the Rete network, and original SQL. It is clear that if non-selective conditions are present, Conditional Rete is superior to the original Rete network.

5 Related Work

TREAT [13] is a variant form of Rete that skips the materialization of intermediate join nodes but joins all nodes in one step. Match Box algorithm [16] pre-computes a rule's binding space, and then has each binding independently monitor working memory for the incremental formation of tuple instantiations. LEAPS [14] is a lazy matching scheme that collapses the space complexity required by Rete to linear.

A generalization of Rete and TREAT, Gator [9] is a non-binary bushy tree instead of a binary bushy tree like Rete, applied to a scalable trigger processing system [10], in which predicate indexing provides computation sharing among common predicates.

The work on Gator networks is more general than ours with respect to employing non-binary discrimination networks with cost model optimizations. However, [9] explores only single tuple updates at a time, does not consider aggregation operators, and is used for trigger condition detection instead of stream processing.

Our work is closely related to several Database research directions, including Active Databases [8][20][18], materialized view maintenance [3], and Stream Database systems. Some recent and undergoing stream projects, STREAM [2], TelegraphCQ [4], and Aurora [1], etc., develop general-purpose Data Stream Management Systems, and focus on general stream processing problems, such as high data rates, bursting data arrivals, and various query output requirements, etc. Compared to these systems, ARGUS tries to solve the performance problem caused by very-large-volume historical data and high data rates by exploiting the very-high-selectivity property of SAMS queries, particularly optimizing incremental query evaluations.

OpenCQ, WebCQ [12], and NiagaraCQ [5] are continuous query systems for Internet databases with incremental query evaluation schemes over data changes. NiagaraCQ's incremental query evaluation is similar to Rete networks. However, it addresses the problem of very large number of queries instead of very-large volume data and high data rates, and the optimization strategy is sharing as much computation as possible among multiple queries. Distinguishably, ARGUS attempts to minimize the intermediate result sizes in both single-query Rete networks and shared multi-query Rete networks. ARGUS applies several techniques such as transitivity inference and conditional materialization toward this goal.

Alert [18] and Tapestry [19] are two early systems built on DBMS platforms. Alert uses triggers to check the query conditions, and modified cursors to fetch satisfied tuples. This method may not be efficient in handling high data rates and the large number of queries in a stream processing scenario. Similar 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). This approach becomes inefficient when the append-only table is very large. Particularly, it has to do repetitive computations over large historical data whenever new data is to be matched in joins.

There is some relevant work on inferring hidden predicates [15][17]. However, they deal with only the simplest case of equi-join predicates without any arithmetic operators. ARGUS deals with general 2-way join predicates with comparison operators and arithmetic operators for Transitivity Inference.

6 Conclusion

Dealing with very-large volume historical data and high data rates presents special challenges for a Stream Anomaly Monitoring System. In ARGUS, Rete networks provide the basic framework for incremental query evaluation, and the very-high-selectivity property of SAMS queries is exploited to minimize intermediate result sizes and speed up performance significantly. The techniques include transitivity inference, user-defined join priority, and conditional materialization. The later two will be replaced by a more general cost-based optimization method that will subsume them. We are also extensively expanding the system to incorporate multi-query computation sharing, incremental aggregation, and the Dynamix Matcher.

Acknowledgements: This work was supported in part by ARDA, NIMD program under contract NMA401-02-C-0033. The views and conclusions are those of the authors, not of the U.S. government or its agencies. We thank Chris Olston for his helpful suggestions and comments, and Bob Frederking, Eugene Fink, Cenk Gazen, Dwight Dietrich, Ganesh Mani, Aaron Goldstein, and Johny Mathew for helpful discussions.

References

1. D. J. Abadi and etc. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
2. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symp. PODS*, 2002.
3. J. A. Blakeley and etc. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. on Database Systems (TODS)*, 14(3):369–400, 1989.
4. S. Chandrasekaran and etc. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*.
5. J. Chen and etc. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proc. of the 18th Intl. Conf. on Data Engineering*, 2002.
6. E. Fink, A. Goldstein, P. Hayes, and J. Carbonell. Search for Approximate Matches in Large Databases. In *Proc. of the 2004 IEEE Intl. Conf. on Systems, Man, and Cybernetics*.
7. C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
8. L. Haas and etc. Startburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):143–160, 1990.
9. E. N. Hanson, S. Bodagala, and U. Chadaga. Optimized Trigger Condition Testing in Ariel Using Gator Networks. Technical Report TR-97-021, CISE Dept., Univ. of Florida, 1997.
10. E. N. Hanson and etc. Scalable Trigger Processing. In *Proc. of the 15th Intl. Conf. on Data Engineering*, 1999.
11. C. Jin and J. Carbonell. ARGUS: Rete + DBMS = Efficient Continuous Profile Matching on Large-Volume Data Streams. Tech. Report CMU-LTI-04-181, Carnegie Mellon Univ. 2004 URL: www.cs.cmu.den/čjin/publications/Rete.pdf.
12. L. Liu, W. Tang, D. Buttler, and C. Pu. Information Monitoring on the Web: A Scalable Solution. *World Wide Web Journal*, 5(4), 2002.
13. D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for IA Production Systems*. Morgan Kaufmann, 1990.
14. D. P. Miranker and D. A. Brant. An algorithmic basis for integrating production systems and large databases. In *Proc. of the Sixth Intl. Conf. on Data Engineering*, 1990.
15. K. Ono and G. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proc. of 16th Intl. Conf. on VLDB*, pages 314–325, 1990.
16. M. W. Perlin. The match box algorithm for parallel production system match. Technical Report CMU-CS-89-163, Carnegie Mellon Univ., 1989.
17. H. Pirahesh and etc. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *Proc. of 13th Intl. Conf. on Data Engineering*, pages 391–400, 1997.
18. U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of 17th Intl. Conf. on VLDB*, 1991.
19. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf.*
20. J. Widom and S. Ceri, editors. *Active Database Systems*. Morgan Kaufmann, 1996.