Optimizing Multiple Continuous Queries

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

Dissertation Committee:
Jaime Carbonell, Carnegie Mellon University (Chair)
Christopher Olston, Carnegie Mellon University, on leave at Yahoo! Research
Jamie Callan, Carnegie Mellon University
Phil Hayes, Vivisimo, Inc.

Summary

October 20, 2006

Abstract

Emerging data stream processing applications present new challenges that are not addressed by traditional DBMS technologies. To provide practical solutions for matching highly dynamic data streams with multiple long-lived and dynamically-updated continuous queries, a stream processing system should support incremental evaluation over new data, query optimization for continuous queries including computation sharing among multiple queries.

This thesis addresses these problems, presents the solutions in a prototype called ARGUS, and conducts experimental evaluations on the implemented techniques. Incremental query evaluation is realized by a set of algorithms based on materializing intermediate results to incrementally evaluate selections/joins (Rete), aggregates (incremental aggregation), and set operators (incremental set operations). The query optimization techniques include transitivity inference to derive highly selective predicates, conditional materialization to selectively materialize intermediate results, join order optimization to reduce join computations, and minimum column projection to project only necessary columns. Computation sharing is realized by an incremental multiple query optimization (IMQO) approach for tractable plan construction and dynamic query registration. It applies four steps to register a new query Q, recording existing query computations of the multi-query plan R, searching common computations between Q and R, selecting optimal sharing paths, and adding new computations to obtain final results for Q and R. The thesis presents a comprehensive computation indexing and searching scheme, and presents several sharing strategies. Finally, the evaluations on two data sets show that each technique leads to significant improvement in system performance up to hundreds-fold speed-up.

ARGUS is implemented atop a widely used commercial DBMS Oracle to allow fast deployment of the prototype as a value-added package to existing database applications where requirements of stream processing are growing rapidly in both scale and diversity.

Future work includes supporting adaptive query processing, supporting distributive and parallel computing, and execution optimization.

Keywords: Stream Data, Continuous Query, Rete, Incremental Query Evaluation, Transitivity Inference, Database, Conditional Materialization, Predicate Set, Extended Predicate Set Operation, Canonical Predicate Form, Predicate Indexing, Computation Sharing.

Contents

1 Introduction

In recent years, we have witnessed the emergence of stream processing applications. The applications include terrorism detection and monitoring from structured message streams, network intrusion detection from NetFlow streams, monitoring wireless sensor network readings in a variety of military and scientific applications, publish/subscribe systems such as stock ticker notification services, and more. In wake of the continuous growth of hardware (network bandwidth, computing power, and data storage) and pervasive computerization into mission-critical tasks, scientific exploration, business, and everyday personal life, stream processing applications has become and will continue to be more attainable, demanding, and prevalent.

These stream processing applications present new challenges that are not addressed by traditional data management techniques, particularly the traditional Database Management System (DBMS) techniques. Two prominent challenges among many are continuous query matching and optimization on large-scale queries. The thesis addresses these two problems with incremental evaluation methods, incremental multiple query optimization, and other related optimization techniques, and implements them into a prototype system ARGUS atop the DBMS Oracle.

Traditional DBMSs do not provide an efficient mechanism to support continuous query matching. While triggers can be defined to simulate the continuous query matching upon data changes, the method is not scalable, the triggers can not be shared, and the functionalities are limited due to the ACID and other design constraints. Moreover, DBMSs do not systematically support efficient continuous matching algorithms that are vital for performance on stream processing. Many query operators can be implemented to efficiently produce new results on new tuples without or with bounded accesses to the historical data, which we call **incremental evaluation**. For example, a selection predicate can be evaluated just on the new tuples without accessing any historical data.

The related work on incremental evaluation is abundant. Various stream join operators were proposed. These include stream joins, such as XJoin [29], MJoin [30], and the window join in [16], and stream aggregates, such as Ripple join [18], window aggregates [21], quantile estimates [13], and top-K queries [4]. These operators are not immediately applicable to ARGUS since it is implemented atop a DBMS. However, when ARGUS migrates to a DSMS, these stream operators are very pertinent, and ARGUS existing incremental evaluation methods should be implemented in the stream operators as well.

Another missing component is the computation sharing module. While multiple query optimization (MQO) [28, 26] has been studied since late 1980's, the techniques are not implemented in commercial DBMS since queries on historical data are typically discarded after a single search. However, since multiple concurrent continuous queries tend to be persistent, computation sharing is appropriate for stream applications and even a must for applications dealing with large-scale queries (pub/sub systems) or with intensive query dynamics (complex and evolving intelligence analysis tasks). Computation sharing can lead to hundreds-fold performance improvement. And the larger the number of concurrently active queries is, the more significant benefit is obtained.

Two problems add complexities to the implementation of a practical sharing component. First, queries arrive intermittently, not in batch, which leads to constant changes of the shared query plan. Second, since global optimization on multiple queries is known to be NP-complete [28, 20], it is impractical to perform one-shot full optimization on large-scale queries, not to say doing it repetitively. A practical solution is to develop a system that adds new queries individually into an existing shared query evaluation plan to obtain reasonably good performance via local optimization. We call this approach **incremental multiple query optimization (IMQO)**. With this approach, the system needs to store existing query computations, identify the common computations between the new query and the existing query plan, choose optimally among multiple sharing paths, and add

unsharable new computations to the plan.

The most distinguishable feature of ARGUS, comparing to other research prototypes [22, 8, 1, 12, 10] on data stream management systems (DSMS), is its comprehensive IMQO framework. While computation sharing has been widely accepted as an important component of a DSMS, it is often missing or underdeveloped in existing research prototypes. For example, STREAM [22, 3, 5, 2] is a general-purpose DSMS prototype developed from scratch by Stanford University with focus on adaptive processing, extended stream query languages, and execution engine architecture. The STREAM architecture is designed to support large-scale concurrent continuous queries and executes a shared query plan that outputs multiple result streams. Algorithms realizing a range of resource sharing strategies are implemented. However, the prototype does not develop or implement the sharing module. It does not recognize the sharable computations across multiple queries, and does not support incremental addition of new queries.

In another example, NiagaraCQ [11, 12] is a publish/subscribe prototype that matches large-scale subscriber queries with Internet content changes. It implements an incremental sharing framework similar to ARGUS. A new query Q can be incrementally added into a shared plan R by identifying sharable computations between Q and R and expanding R with new unsharable computations. However, NiagaraCQ applies a simplified approach to identify sharable computations, i.e. exact string match and shallow syntactic analysis to identify equivalent and subsumption predicates. This largely limits the potential improvement offered by computation sharing. In NiagaraCQ, simple selection predicates are grouped by their expression signatures and evaluated in chains, and equi-join predicates can also be shared. But the identification is limited to the predicate level. Such limited findings restrict the construction of efficient shared plans. For example, it can not identify sharable nodes which present the results of a group of predicates (PredSet). Without topological association, such findings also limit the sharing strategies to be applied.

This thesis addresses incremental evaluation, IMQO, and several query optimization techniques pertinent to continuous queries. The resulting techniques are implemented in the ARGUS prototype. ARGUS supports continuous large-scale complex query matching. The shared query plan has the persistent storage and can be incrementally expanded with new queries.

ARGUS is built atop a commercial DBMS Oracle. Such choice allows us to focus on the stream-related problems without worrying about implementing the underlying execution engine. A more interesting and instant benefit of using DBMS is that ARGUS can be offered as a value-added package to existing database applications where the requirement of stream processing is emerging. We are looking for opportunities to apply the work in national geo-spatial databases, for instance. While the DBMS is the current choice, our ultimate goal is to make the techniques in practical DSMS systems as well as they also reach a state of maturity.

1.1 Thesis Statement and Contributions

The thesis statement is:

> The thesis demonstrates constructively that incremental multiple query optimization, incremental query evaluation, and other query optimization techniques provide very significant performance improvements for large-scale continuous queries, and are practical for real-world applications by permitting on-demand new-query addition. The methods can function atop existing DBMS systems for maximal modularity and direct practical utility. And the methods work well across diverse applications.

The thesis contributions are as following.

- **Incremental multiple query optimization:** We design, implement, and evaluate an IMQO framework. It is comprised of the following components.

  - A comprehensive computation indexing scheme to support general plan structures for selection-join-projection queries, aggregate queries, and set operation queries.
  - A set of efficient algorithms to search common computations.
  - Several effective sharing strategies to construct shared query network.
  - A set of tools to incrementally construct the query network and to update the computation index.

- **Incremental evaluation:** We design, implement, and evaluate the incremental evaluation mechanism for selection, join, algebraic aggregates, and set operations.
- **Query optimization:** We explore and evaluate several effective query optimization techniques, including join order optimization, conditional materialization, transitivity inference, and minimum column projection.
- **System implementation:** We build the system atop a DSMS Oracle for direct practical utility for existing database applications where the needs of stream processing become increasingly demanding.
- **Evaluation:** We study and analyze the effectiveness of the implemented techniques. It shows that each technique provides significant performance improvements for general or specific queries, and that the implemented system supports large-scale continuous queries efficiently across different applications.

In this summary, Section 2 overviews the ARGUS system architecture, Section 3 describes the incremental evaluation, Section 4 describes the query optimization techniques, Section 5 describes the incremental multiple query optimization, Section 6 describes the code assembly, Section 7 presents partial evaluation results, and Section 8 summarizes the contributions and points to future work.

2 System Overview

The thesis implements the ARGUS stream processing system. This prototype is a part of the large project ARGUS sponsored by DTO NIMD program and jointly managed by Carnegie Mellon University and Dynamix Technologies Inc.

As part of the ARGUS project, the ARGUS stream processing system is implemented atop Oracle DBMS for immediate practical utility. The system takes continuous queries specified in SQL, generates shared query evaluation plans, and evaluates plans against stream data.

ARGUS contains two components, Query Network Generator (NetGen) and Execution Engine (Engine), shown in Figure 1. The system works as following. An analyst sends a request to ARGUS to register a new query Q; NetGen analyzes the query, constructs a new shared optimal query evaluation plan (also called query network) from the existing one, instantiates the plan and generates the updated initialization and execution code, records the plan information in the system catalog, and outputs the updated code; Engine runs the query network to match with data streams and returns the results to the analyst.

2.1 Execution Engine

The engine is the underlying DBMS query execution engine. We use its primitive relation-operator support, but not its complex query optimization functionality, to evaluate the query network generated by ARGUS to produce stream results. As we know, to run a query in SQL, a DBMS generates an optimal logical evaluation plan, then instantiates it to a physical plan, and executes the physical plan to produce the query results. The logical plan can be viewed as a formula comprised of relation operators on the querying relations. And the physical plan specifies the actual methods and the procedure to access the data. When the query is simple, e.g. a selection or an aggregate from one relation, or a 2-way join or a UNION of two relations, the logical plan is simple and requires almost no

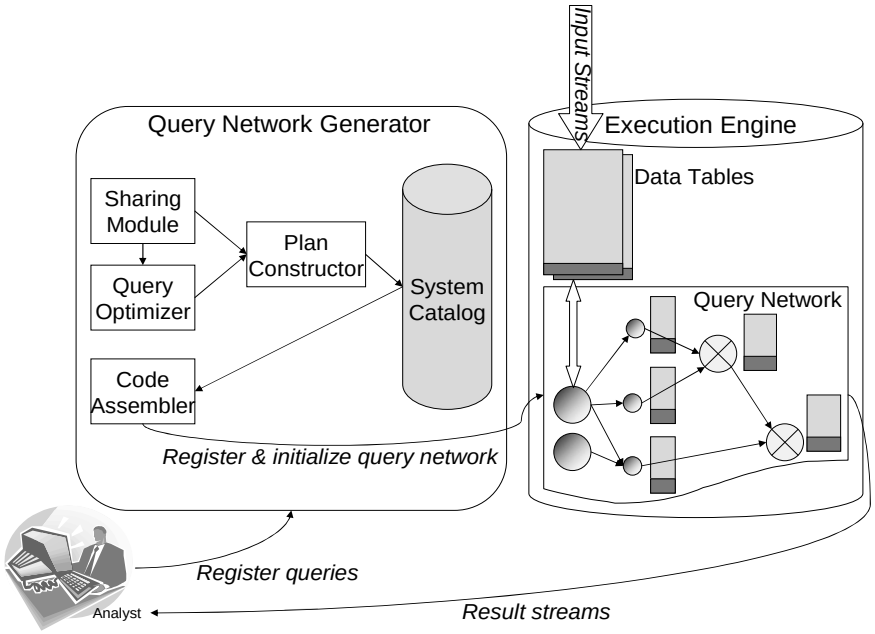

Figure 1: Using ARGUS. An analyst registers a query with ARGUS. ARGUS Query Network Generator processes and records the query in the System Catalog, and generates the initialization and execution code. ARGUS Execution Engine executes the query network to monitor the input streams, and returns matched results. The analyst may register more queries.

effort from the query optimizer. An ARGUS query network breaks the multiple complex continuous queries into simple queries, and the DBMS runs these simple queries to produce the desired query results. Therefore, in ARGUS, the underlying DBMS is not responsible for optimizing the complex logical plans, but is responsible for optimizing and executing physical plans for the simple queries.

2.2 Query Network Structure

A query network is a directed acyclic graph (DAG). Figure 2 shows an example, which evaluates four queries. The upper part evaluates two sharable selection-join queries, and the lower part evaluates two sharable aggregate-then-join queries. The source nodes (nodes without incoming edges) present original data streams and non-source nodes present in-

termediate or final results.

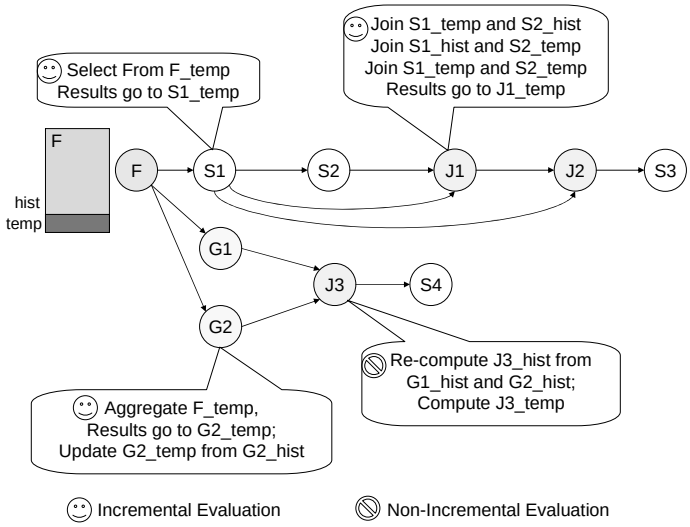

Figure 2: Execution of a shared query network. Each node has a historical (hist) table and a temporary (temp) table, here only those of node F are shown. The callouts show the computations performed to obtain the new results that will be stored in the nodes' temporary tables. S nodes are selection nodes, J nodes are join nodes, and G nodes are aggregate nodes. The network contains two 3-way self-join queries and two aggregate-then-join queries, and nodes J2, S3, J3, and S4 present their results respectively.

Each network node is associated with two tables of the same schema. One, called **historical table**, stores the historical data (original stream data for source nodes, and intermediate/final results for non-source nodes); and the other, called **temporary table**, temporarily stores the new data or results which will be flushed and appended to the historical table later. These tables are DBMS tables, their storage and access are controlled by the DBMS.

In principle, we are only interested in the temporary data because it presents the new query results. However, since some operators, such as joins and aggregates, have to visit the history to produce new results, the historical data has to be retained too. It is possible to retain only certain nodes' historical data that will be accessed later. However, this has more intricacy when sharing is involved, and is not supported in current implementation.

An arrow between nodes presents the evaluation of a set of operators on the parent

node(s) to obtain the results stored in the child node. The operator sets are stream operator sets. They operate on streams and output other streams. Many operator sets can be evaluated incrementally on the parents' temporary tables to produce new results that populate the result node's temporary table, while some others can not. Regardless a node can be incrementally evaluated or not, the way to populate its temporary table can be expressed by a set of simple SQL queries operating on its parent nodes and/or its own historical table. In another word, the incremental or non-incremental evaluation methods to populate a node's temporary table can be instantiated by simple SQL queries.

Each node is associated with two pieces of code and a runtime Boolean flag. The first code, **initialization code**, is a set of DDL statements to create and initialize the historical and temporary tables. It is executed only once prior to the continuous execution of the query network. The second, **execution code**, is a PL/SQL code block that contains the simple queries to populate the temporary table. The Boolean flag is set to true if new results are produced. To avoid fruitless executions, the queries are executed conditioning on the new data arrivals in the parent nodes. Particularly, only when at least one parent flag is true, are the queries executed. There is a finer tuning on execution conditions for incremental joins depending on which parent's temporary table is used.

The nodes of the entire query network are sorted into a list by the code assembler. Correspondingly, we get a list of execution code blocks. This list of code blocks are wrapped in a set of Oracle stored procedures. These stored procedures are the execution code of the entire query network. To register the query network, the system runs the initialization codes, then stores and compiles the execution code. Then the execution code is scheduled periodical executions to produce new results.

3 Incremental Evaluation

In a stream processing system, new data tuples continuously arrive, and long-lived queries continuously match them to produce new results. Efficient algorithms (**Incremental Evaluation**) to produce new results on new tuples with minimal access to the historical data is vital for performance.

We implemented efficient incremental evaluation algorithms for selection, join, algebraic aggregates, and set operators (union, union all, and set difference).

Selection is easy. New results can be produced without access to historical data.

Join is more complex. A new result may be produced from the join of a new tuple with old tuples in the history or the join of new tuples, but will never be produced from the join of only old tuples. We implemented the incremental evaluation algorithm for 2-way joins by performing the two types of small joins: the join between the new data parts and the join between the new data part versus old data part.

The incremental selection and join methods were inspired from the Rete algorithm [15] which stores intermediate results to save repetitive computations in recursive matching of newly produced working elements.

Many aggregate functions, *algebraic functions*, including MIN, MAX, COUNT, SUM, AVERAGE, STDDEV, and TrackClusterCenters, can be incrementally updated upon data changes without revisiting the entire history of grouping elements (incremental aggregation); while other aggregates, *holistic functions*, e.g. quantiles, MODE, and RANK, can not be done this way. Particularly, algebraic functions can be updated upon data changes from bounded statistics, while holistic functions can not. For example, AVERAGE can be updated from up-to-date SUM and COUNT statistics which are simple accumulations upon data changes. We implemented incremental aggregation for general algebraic aggregates including arbitrary user-defined ones.

We implemented incremental evaluation for union, union all, and minus (set difference)

operators.

4 Query Optimization

We studied and implemented several query optimization techniques in ARGUS. These include transitivity inference, join ordering, conditional selection materialization, and minimum column projection.

Transitivity inference derives implicit highly-selective predicates from existing query predicates to filter out many non-result records in earlier stages and reduce the amount of data to be processed later. For example, assume a query has following conditions (the first is very selective): $r1.amount > 1000000$, $r2.amount > r1.amount * 0.5$, and $r3.amount = r2.amount$. The first two predicates imply a new selective predicate on $r2$: $r2.amount > 500000$. Further, the third predicate and the newly derived predicate imply another new selective predicate on $r3$: $r3.amount > 500000$. These inferred predicates have significant impact on performance. The intermediate result tables of the highly-selective selection predicates are very small and save significant computation on subsequent joins. The experiments show up to twenty fold improvement for applicable queries. There are relevant works on inferring hidden predicates [23, 24]. However, they deal with only the simplest case of equijoin predicates without any arithmetic operators. With our canonicalization procedure, ARGUS is able to derive implicit selection predicates from general 2-way join predicates and other selection predicates.

Searching for the optimal join order is one important goal for traditional query optimizers. The optimal join order can lead to hundreds-fold faster plans than non-optimal ones. This problem is still pertinent to continuous queries. We implemented the optimizer to search for the optimal join sequence by using historical data for estimating costs.

In the incremental evaluation of selections/joins, materialized intermediate results improve performance by avoiding repetitive computations over the historical data. However,

a potential problem is that when any materialized intermediate table is very large, thus requiring many I/O operations, the performance degrades severely. When intermediate results are not reduced substantially from the original data, the time saved from the repetitive computations may be offset or exceeded by the materialization overhead (I/O time). Conditional materialization examines the selectivity of selection PredSets and decides whether or not materializing the PredSets based on a threshold cutoff (default is 0.3). Conditional materialization is implemented in the query optimizer. Conditional materialization shows up to 1.8-fold performance improvement in our experiments

Minimum column projection refers to projecting the minimal set of columns for intermediate tables. When a new node is created, to save materialization space and execution time, we only project the necessary columns from its parents. These columns include those in the final results and those needed for further evaluation. The process becomes intricate when sharing is considered. When a node is shared among computations of different queries, it may not contain all the columns needed for the new query. Then extra columns will be added to the node and possibly to its ancestors. This process is called *projection enrichment*. Aurora [6] has the same functionality to project minimum columns. However, with its procedural query language, the sharing-related intricacy is not considered by the system but is handled manually.

5 Incremental multiple query optimization (IMQO)

A generally useful IMQO framework must support extensive query types and general plan structures efficiently. Particularly, the framework should meet following requirements.

- Support general query types, e.g. selection-join-projection queries, aggregate queries, set operation queries, and their combinations.
- Support general plan structures, e.g. materializing the results of grouped predicates.

- Support compact indexing storage, fast computation search, and easy index update and plan expansion for large-scale query applications.

Each requirement presents specific challenges, and the combination leads to a hyper-linear complexity growth. Procedurally, IMQO involves four complex steps:

- computation indexing,
- common computation identification,
- sharing path selection,
- and indexing update and plan expansion.

Each step interacts with others and presents specific problems. So the framework should be designed systematically to meet the overall requirements, as well as address the specific problems in each step.

5.1 Indexing and Searching

The first two steps of IMQO, indexing and identifying common computations, are essential to construct efficient shared plans, since the identification capability directly determines to what extent the sharing can be achieved, determines what heuristic sharing strategies can be applied, and largely influences applicable shared plan structures.

Common subexpression identification is also known to be NP-hard [19, 25], and thus has to be addressed heuristically. Secondly, identifying sharable computations from the shared plan structures and topologies, not just from the query semantics, adds one more layer of complexity. Thirdly, to scale to a large number of queries, the scheme and algorithms should support compact index storage, fast index search and easy index update. And finally, since the shared plan dynamically evolves as new queries are registered, and is subject to local re-optimization to adapt to data distribution changes, the indexing scheme

should provide enough information for fast constructing, updating, and rearranging the executable query evaluation plan.

Previous work on DSMSs, such as NiagaraCQ, STREAM, and other work described, either do not develop or underdevelop the support of the common computation indexing and identification. Previous work on MQO [27, 14, 7, 9] and view-based query optimization [17, 31] address the common computation identification problem specifically, but do not concern with plan topologies, compact storage, and easy update, and use quite different approaches from this thesis.

This thesis introduces a comprehensive computation indexing scheme and related searching algorithms to index and search common computations. Particularly, it emphasizes the identification capability, and the solution to large-scale queries. It provides a general systematic framework to index, search, and present common computations, done to a degree well beyond the previous approaches.

In terms of identification capability, the scheme indexes and identifies sharable selection nodes, join nodes, aggregate nodes, and set operator nodes. To identify sharable selections and joins, the scheme recognizes syntactically-different yet semantically-equivalent predicates and expressions by canonicalization, and subsumptions between predicates and predicate sets, and supports self-join which is neglected in previous work. It supports rich predicate syntax by indexing predicates in CNF forms, and it supports fast search and update by indexing multiple plan topology connections. To identify sharable aggregate nodes and sharable set operator nodes, the scheme recognizes the subsets and supersets of GROUPBY expressions, and the subsets and supersets of set operator tables.

To deal with the large-scale problem, the scheme applies a relational model, instead of a linked data structure as by previous approaches. All the plan information is stored in the system catalog, a set of system relations. The advantage is that the relational model is well supported by DBMSs. Particularly, fast search and easy update are achieved by

DBMS indexing techniques, and compact storage is achieved by following the database design methodologies.

The computations of a query network is organized as a 4-layer hierarchy. From top to bottom, the layers are topology layer, PredSet layer, OR predicate (ORPred) layer, and literal predicate (literal) layer. The last three layers, also referred as the *three-pred layers*, present the computations in CNF. And the top layer presents network topological connections.

Such a hierarchy supports general predicate semantics and general topology structures. An indexing scheme should efficiently index all relevant information of the hierarchy to support efficient operations on it including search and update.

The reason that we do not use a linked data structure to record query network is due to its search and update inefficiency on large-scale query networks. In a linked data structure, the update needs to perform the search first unless the nodes are indexed, and the search needs to go through every node of the same querying table(s) to check the relationship between the node's associated operator set and the query's operator sets to decide the sharability.

There are several issues we need to consider before we transform the computation hierarchy to the relational model. Particularly, we want to deal with rich predicate syntax for matching semantically-equivalent literal predicates, match self-join computations at the three-pred layers, identify subsumptions at the three-pred layers, and identify complex topological connections.

5.1.1 Rich Syntax and Canonicalization

A literal predicate can be expressed in different ways. To match them quickly and accurately, we introduce a canonicalization procedure. It transforms syntactically-different yet semantically-equivalent literal predicates into the same pre-defined canonical form. Then

the equivalence can be detected by exact string match.

There is intricacy with regard to the canonicalization. We need to identify subsumption relationship between literal predicates. For example, $t1.a > 10$ subsumes $t1.a \geq 5$. The exact match on the canonicalized predicates can not identify subsumptions. Instead, the subsumption can be identified by a combination of the exact match on the column references, the operator comparison, and the constant comparison.

5.1.2 Self-Join

True table names should be used in canonical forms to conduct the fast exact-string matching. However, this is not practical for a self-join predicate. The specification of joining two records is clarified by different table aliases. To retain the semantics of the self-join, we can not replace the table aliases with their true table names. To avoid the ambiguity or information loss, we introduce Standard Table Aliases (STA) to reference the tables. We assign $T1$ to one table alias and $T2$ to the other. To support multi-way join predicates, we can use $T3$, $T4$, and so on [1]. Then STA assignments are enumerated for a new predicate p to search its matches in the system catalog.

Self-joins also present problems in the middle layers (PredSet and ORPred layers). For example, an ORPred p_1 may contain two literal predicates, one is a selection predicate ρ_1: $F.c = 1000$, and the other a self-join predicate ρ_2: $T1.a = T2.b$. Therefore, ρ_1's STA, $T1$ or $T2$, must be indexed in p_1. Similar situation exists in PredSets where some ORPred is a selection from a single table and some other is a self-join.

5.1.3 Subsumption at Middle Layers

Given an ORPred p of a PredSet P in the new query Q, we want to find all ORPreds $p' \in R_{ORPred}$, such that p is subsumed by, subsumes, or is equivalent to p', based on the subsumptions identified at the literal layer. From the results, we find all PredSets $P' \in R$,

[1] Multi-join is not supported by ARGUS currently

such that P is subsumed by, subsumes, or is equivalent to P'. These search algorithms are implemented in the system

5.1.4 Topology Connections

PredSets are associated with nodes. A PredSet P presents the topological connection between the associated node N and N's ancestors $\{A\}$. A node N is associated with multiple PredSets depending on the different types of ancestors. An important one is the DPredSet which connects N to its direct parents. DPredSet is used in constructing the execution code, and needs to be indexed.

Solely relying on DPredSets causes chained search on branches for both selection and join PredSets. To address the problem, we record two more PredSets for each node N. These PredSets are associated with nodes called N's SVOA and N's JVOAs.

A selection node N's SVOA is N's closest ancestor node that is either a join node or a base stream node. A join node or a base stream node N's SVOA is itself. SVOA stands for selection very original ancestor. A join node N's JVOAs are the closest ancestor nodes that are either join nodes (but not N) or base stream nodes. A selection node N's JVOAs are the JVOAs of N's SVOA. And a base stream node's JVOA is NULL. JVOA stands for join very original ancestor.

SVOAs present local selection chains, and JVOAs present one join depth beyond the local selection chains. With SVOAs and JVOAs, the chained searches are no longer necessary.

5.1.5 Relational Model for Indexing

Considering all the issues and the 4-layer hierarchy, we designed the relational model for indexing. Two adjustments are made. First, the relations that index literal predicates and ORPreds are merged into one, *PredIndex*, based on the assumption that ORPred are not

frequent in queries. This allows a literal predicate to appear multiple times in *PredIndex* if it belongs to different ORPreds. But this redundancy is negligible given the assumption. The second adjustment is splitting the node topology indexing relation (Node Entity in the ER model) to two, namely, *SelectionTopology*, and *JoinTopology*, based on the observation that the topology connections on selection nodes and on join nodes are quite different.

5.2 Sharing Strategies

Given the sharable nodes identified, various sharing optimization strategies may be applied. We present two simple strategies, match-plan and sharing-selection. Match-plan matches the plan optimized for the single new query with the existing query network from bottom-up. This strategy may fail to identify certain sharable computations by fixing the sharing path to the pre-optimized plan. Sharing-selection identifies sharable nodes and chooses the optimal sharing path.

In both strategies, we need to choose the optimal sharable node at each JoinLevel. We apply a simple cost model for doing so. The cost of sharing a node S is simply the cost of evaluating the remaining part of the chosen PredSet P. The cost is defined as the size of S, the number of records to be processed to obtain the final results of P. When a join node J is chosen for sharing, even if it does not provide the final results for the chosen join PredSet P, we choose not to extend J for P until it is the last join in the query. Instead the remaining computations are rewritten as a selection PredSet from J, and thus are carried on to the next JoinLevel. With this sharing choice, we are able to create less join nodes. This choice is applied by both sharing-selection and match-plan.

5.3 Incremental Multiple Query Optimization on Aggregates

Algebraic functions can also be shared through dimension reductions, or *vertical expansion*. But holistic functions can not. In the vertical expansion, a new aggregate node N is created

from an existing node G and further groups the results of G.

Similar to the IMQO on selection and join queries, the IMQO on aggregate queries works as following for a new query B:

- Identify sharable nodes $\{A\}$ with following steps:
  - identify all dimension sets $\{\mathbb{D}_{A'}\}$ that are supersets of $\mathbb{D}_B$ by looking at *GroupExprSet* and *GroupExprIndex*,
  - identify the nodes $\{A'\}$ associated with $\{\mathbb{D}_A\}$ by looking at *GroupToplogy*,
  - and identify the nodes in $\{A'\}$ that contain all columns needed for query B. These are sharable nodes $\{A\}$.
- Select the optimal node A from which B will be evaluated.
- Create a new node B by creating the table pairs and updating the system catalog.
- Perform rerouting on B.

Given the new query B, there may be multiple nodes from which a vertical expansion can be performed. According to the time complexity analysis, the optimal choice is the node A such that $|A_H|$ is the smallest. If A does not contain all aggregate functions or bookkeeping statistics needed by query B, a *horizontal expansion* is performed.

After the new node B is created, the system invokes the rerouting procedure. It checks if any existing node C can be sped up by being evaluated from B. We apply a simple cost model to decide such rerouting nodes. If 1. B contains all the aggregate functions needed by C, and 2. $|B_H| < |P^C_H|$ where P^C is C's current parent node, then C will be rerouted to B. The system applies a simple pruning heuristic. If a node C satisfies both conditions, and a set of nodes $\{C_i\}$ satisfying the first condition are descendants of C, then any node in $\{C_i\}$ should not be rerouted, and so are dropped from consideration.

The evaluation shows up to hundred-fold performance improvement over non-sharing approaches.

5.4 Incremental Multiple Query Optimization on Set Operators

The IMQO on set operators is similar to those on selection/join queries and aggregate queries. A set operator node is the result of a set operation on two or more nodes. A set operator node can be shared if it provides the data from which final results can be computed.

Given a new set operator query Q that operates on a set of tables $\{M_v\}$, we want to find a set of set operator nodes $\{N\}$ from the existing query network R where N can be used to evaluate Q. The sharability depends on the set operators, the operation tables, and the columns. For example, when Q is UNION, a sharable N should be either UNION or UNION ALL; and Q's operation table set is a superset of N's operation table set. For a sharable N, Q's column set must be a subset of N's column set, and their column position mapping across the operation tables must be consistent.

Once the sharable nodes $\{N_v\}$ are identified, we want to choose the optimal sharable node N. N is chosen as following.

- When Q is UNION ALL, choose N whose table size is maximum.
- When Q is UNION, choose N whose number of distinct values on Q's requested columns is the maximum.
- When Q is MINUS, if there are Ns that are MINUS, choose the N whose table size is minimum; otherwise, if there are sharable Ns that are UNION/UNION ALL, choose the N whose number of distinct values on Qs requested columns is the maximum.

6 Code Assembly

The query network is evaluated in a linear fashion, and the nodes need to be sorted. The only sorting constraint is that the descendant nodes must follow their ancestor nodes.

One way to get a valid order is to traverse the entire network starting from the original stream nodes. However, this entails many system catalog accesses and is not efficient. On the other hand, a predefined one-dimensional sorting order is too rigid to query network updates.

We introduce a two-dimensional sort ID assignment scheme. A sort ID is a pair of integers, JoinLevel and SequenceID. The JoinLevel globally defines the depth of a node, and the SequenceID defines the order within the local area of the same join depth nodes. In a query network of only selection and join nodes, a node's JoinLevel is its join depth. An original stream node's JoinLevel is 0. For a node with two or more parents, a.k.a. a join node or a set operator node, its JoinLevel is 1 plus the maximal JoinLevel of its parents. For a node with a single parent, a.k.a. a selection node or an aggregate node, its JoinLevel is the same to its parent JoinLevel.

When a new node N is created as a leaf node of the query network, its SequenceID is assigned as k plus its parent's SequenceID. In the system, the default is $k = 1000$. When a node is inserted into between a parent node and a child node in a local tree, the new node's SequenceID is the round-up mean of its parent and child's SequenceIDs. So a large k helps future insertions without affecting children's SequenceIDs. If the parent and child's SequenceIDs are consecutive, and thus no unique SequenceID in-between is available for the new one, then the system increments the SequenceIDs of the child and all its descendants in the local tree by k.

With JoinLevel and SequenceID defined, a valid order can be obtained by sorting on the JoinLevels and then on the SequenceIDs.

7 Evaluation

We conduct experiments to understand the effectiveness of various techniques implemented in ARGUS. Particularly, we evaluate the effectiveness of incremental evaluation methods and optimization techniques on selection-join-projection queries, the effectiveness of incremental aggregation and IMQO on aggregate queries, and the effectiveness of IMQO techniques on SJP queries. The evaluated IMQO techniques include canonicalization, join sharing, and the two sharing strategies, match-plan and sharing-selection. The results show that every individual technique lead to significant performance improvement either in general or at least for some specific types of queries. As a whole, the system provides acceptable performance for continuously matching large-scale queries. The analysis of the results also raises new questions and points to new research directions.

In this summary, we only show the results on IMQO techniques on SJP queries.

7.1 Experiment Setting

We use two databases, the synthesized FedWire money transfer transaction database (Fed), and the anonymized Massachusetts hospital patient admission and discharge record database (Med). Both databases have a single stream with timestamp attributes.

This summary only shows the results on Fed. Fed is a synthesized database containing 500006 FedWire money transfer transactions. The schema contains all the real transaction data attributes. The data are generated according to the real transaction statistics.

We created a query repository for Fed and Med databases. These queries are generated systematically. First, interesting queries arising from applications are formulated manually as query seeds or query categories. The seeds cover a wide range of query types, including selections, joins, aggregates, set operators, and their combinations. The seed queries vary in several ways and present some overlap computations. For example, there are 2-way, 3-way, 4-way, and 5-way self-join queries, and later may share from the results of the former

ones. The queries generated from the same query seed present overlap computations that can be identified as subsumptions and be shared.

The experiments were conducted on an HP PC computer with single core Pentium(R) 4 CPU 3.00GHz and 1G RAM, running Windows XP. To simulate the streams, in the order of time, we take the first part (300000 records from Fed and 600000 from Med) of the data as historical data, and simulate the arrivals of new data incrementally.

7.2 Incremental Multiple Query Optimization on SJP Queries

We compare performance of four query network generation configurations, AllSharing, NonJoinS, NonCanon, and MatchPlan, as shown in Table 1. Particularly, we conduct three comparisons: 1. join sharing vs. non-join sharing, i.e. AllSharing vs. NonJoinS; 2. canonicalization vs. non-canonicalization, i.e. AllSharing vs. NonCanon; and 3. sharing-selection vs. match-plan, i.e. AllSharing vs. MatchPlan. Figures 3 show the times to evaluate multiple queries scaling from 100 queries to 768 queries for Fed. When comparing sharing-selection and match-plan, we also present a baseline curve for the configuration of match-plan without canonicalization (MatchPlan_NCanon), which simulates NiagaraCQ's approach.

Config ID	Join Sharing	Canoni-calize	Strategy
AllSharing	Y	Y	Sharing-Selection
NonJoinS	N	Y	Sharing-Selection
NonCanon	Y	N	Sharing-Selection
MatchPlan	Y	Y	Match-Plan

Table 1: Network Generation Configurations. Functionality enabled: Y; disabled: N.

As shown in Figure 3, the performance difference between join sharing and non-join sharing is significant. This is because sheer repetitive join work is computed multiple times for non-join sharing.

As shown in Figure 3, the effect of canonicalization is also significant, particularly

on Fed, due to different query characteristics. In Fed queries, there is a significant portion of queries that specify different time windows for join, such as $r2.tran\_date <= r1.tran\_date + 20$ and $r2.tran\_date <= r1.tran\_date + 10$. The canonicalization procedure makes it possible to identify the subsumption relations between such join predicates. Thus the sharing leads to more significant reduction in the number of join nodes.

In Figures 4, we compare sharing-selection, match-plan, and match-plan without canonicalization. It is not surprising that match-plan without canonicalization is worse than the other two because of the effect of canonicalization. When both perform canonicalization, sharing-selection is still better than match-plan by identifying more sharing opportunities and constructs smaller query networks.

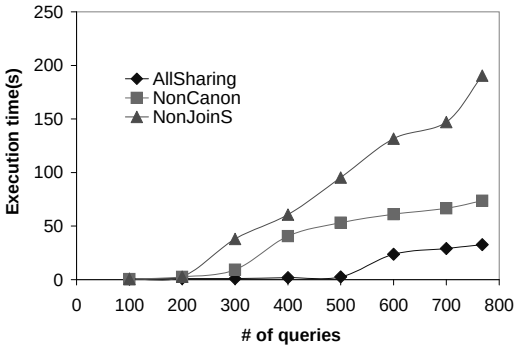

Figure 3: Fed Canonicalization. This shows the effectiveness of canonicalization. The canonicalized query networks are up to double performance improvement.

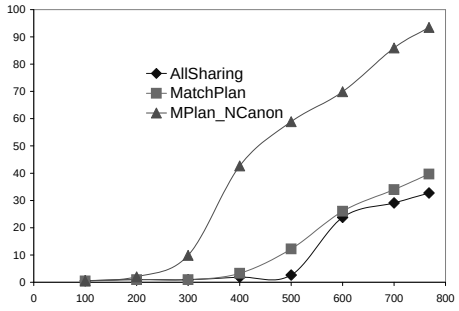

Figure 4: Fed match-plan vs. sharing-selection. This compares the total execution times of query networks generated with sharing-selection (AllSharing), match-plan (MatchPlan), and match-plan without canonicalization (MPlan_NCanon).

8 Conclusion and Future Work

This section first overview the future work, and then summarize the thesis contributions.

For the future work, we consider extending the current work, and adding new capabilities, particularly adding adaptive processing functionalities.

To extend the current work, we consider supporting multi-way joins and more sophisticated local re-optimization techniques including restructure sharing and rerouting.

Adaptive query processing (AQP) has received much attention recently in the stream processing research. Equipped with a comprehensive computation indexing scheme that provides a clear view of the current plan and fast search and update tools, the system can easily apply re-optimization strategies, such as adaptive plan restructuring and rerouting. To support dynamic plan re-optimization, the system needs to look at the local plan region, and reconstruct the new local plan based on the new cost estimates.

Dynamic operator scheduling is another important adaptive processing technique. Query network nodes are evaluated sequentially. The evaluation order only needs to satisfy one constraint: a node must be evaluated before any of its descendants. Among multiple valid orders, the optimal one that minimizes disk page swapping may change over time. Dynamic rescheduling aims to identify the needs of the rescheduling and efficiently reorganize the execution order.

8.1 Summary of Contributions

The thesis addresses the challenges of continuously matching a large number of concurrent queries over high data-rate streams and it is specifically targeted at detecting rare high-value "hits" such as alert conditions.

In order to provide practical solutions for matching highly-dynamic data streams with multiple long-lived continuous queries, the stream processing system supports incremental evaluation, query optimization for continuous queries, and incremental multiple query optimization.

The thesis demonstrated constructively that incremental multiple query optimization, incremental query evaluation, and other query optimization techniques provide very significant performance improvements for large-scale continuous queries, and are practical for

real-world applications by permitting on-demand new-query addition. The methods can function atop existing DBMS systems for maximal modularity and direct practical utility. And the methods work well across diverse applications.

We implement a complete IMQO framework that supports large-scale general queries including selection-join-projection queries, aggregate queries, set operator queries, and their combinations. It provides a practical solution to large-scale queries and allows on-demand query addition, a requirement in many real applications.

In the center of the IMQO framework are the comprehensive computation indexing scheme and the related common computation search algorithms realized by the relational model for compact storage, fast search, and easy update. This approach is much more advanced and general than previous work done for DSMSs, in terms of supporting more types of queries, supporting more flexible plan structures, and identifying more general types of common computations. The approach is also very different from previous work done for MQO and VQO which usually employ query graphs and do not index plan topologies. And the approach is efficient in time since it searches only the relevant computations and formulates the conceptual common computations in a bottom-up fashion.

There are several computation description issues relevant to common computation identification, including semantically-equivalent yet syntactically-different predicates and expressions, self-join presentations, subsumption identification, predicates with disjunctions, and plan topology presentations. The intertwined nature of the problems add much more complexity to the scheme design and algorithm development. We apply various techniques and solve the problems in the integrated scheme design. These include the 4-layer hierarchical indexing model, predicate and expression canonicalization, triple-string canonical form, standard table alias presentation and search at multiple layers, subsumption identification at multiple layers, and multiple topology presentations.

The IMQO framework applies several sharing strategies to construct shared query net-

works that result in up to hundreds of time fold improvement comparing to unshared ones. These include the match-plan and sharing-selection for selection-join-projection queries, aggregate-sharing-selection and aggregate-rerouting for aggregate queries, and set-operator-sharing for set operator queries. Sharing-selection usually results in more compact query networks.

The thesis implements the incremental evaluation methods for selection, join, algebraic aggregates, and set operators.

The thesis implements several effective query optimization techniques, including transitivity inference for inferring highly-selective predicates, conditional materialization for selectively materializing intermediate results, join order optimization for reducing join computation, and minimum column projection for projecting only necessary columns.

The system is built atop a DSMS Oracle for direct practical utility for existing database applications where the needs of stream processing become increasingly demanding.

Finally, the evaluations show that every individual technique leads to significant improvement in system performance up to hundreds fold speed-up.

References

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

[2] 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.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[4] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *SIGMOD Conference*, pages 28–39, 2003.

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

[6] Donald Carney, Ugur Çetintemel, 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 *VLDB*, pages 215–226, 2002.

[7] Upen S. Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986.

[8] 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 *CIDR*, January, 2003.

[9] Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Trans. Knowl. Data Eng.*, 10(3):493–499, 1998.

[10] Jianjun Chen and David J. Dewitt. Dynamic Re-grouping of Continuous Queries. Technical Report 507, CS, University of Wisconsin-Madison, 2002.

[11] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[12] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[13] Graham Cormode, Minos N. Garofalakis, S. Muthukrishnan, and Rajeev Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD Conference*, pages 25–36, 2005.

[14] Sheldon J. Finkelstein. Common subexpression analysis in database applications. In *SIGMOD Conference*, pages 235–245, 1982.

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

[16] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[17] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, 2001.

[18] 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.

[19] Matthias Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, pages 191–205. Springer, 1985.

[20] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[21] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conf*, pages 311–322, 2005.

[22] 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.

[23] Kiyoshi Ono and Guy Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Australia, 1990.

[24] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *Proceedings of the 13th International Conference on Data Engineering*, pages 391–400, Birmingham, U.K., April, 1997.

[25] Daniel J. Rosenkrantz and Harry B. Hunt III. Processing conjunctive predicates and queries. In *VLDB*, pages 64–72, 1980.

[26] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference*, pages 249–260, 2000.

[27] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[28] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.

[29] 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.

[30] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[31] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD Conference*, pages 105–116, 2000.