2006

# ARGUS: Efficient Scalable Continuous Query Optimization for Large-Volume Data Streams

Chun Jin
*Carnegie Mellon University*

Jaime G. Carbonell
*Carnegie Mellon University*

# ARGUS: Efficient Scalable Continuous Query Optimization for Large-Volume Data Streams

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

**Abstract**

We present the methods and architecture of ARGUS, a stream processing system implemented atop commercial DBMSs to support large-scale complex continuous queries over data streams. ARGUS supports incremental operator evaluations and incremental multi-query plan optimization as new queries arrive. The latter is done to a degree well beyond the previous state-of-the-art via a suite of techniques such as query-algebra canonicalization, indexing, and searching, and topological query network optimization. Building on top of a DBMS, the system provides a value-adding package to the existing database applications where the needs of stream processing become increasingly demanding. Compared to directly running the continuous queries on the DBMS, ARGUS achieves well over a 100-fold improvement in performance.

## 1 Introduction

In wake of the continuous growth of hardware (network bandwidth, computing power, and data storage) and pervasive computerization of business and personal lives, the need of data stream processing becomes increasingly demanding. It readily arises from the existing DBMS applications to manage the increasing data dynamics, and presents new challenges that are not addressed by traditional DBMS technologies. The two of the prominent challenges are to match continuous queries efficiently, and to optimize multiple concurrent queries effectively. Along with other challenges has led to the emergence of Data Stream Management Systems (DSMS), either as integral components of a DBMS (this paper) or as stand-alone alternatives (most other work).

Efficient continuous query matching requires incrementally evaluating persistent query operators. A traditional DBMS operator operates on relations and the output is another relation. In contrast, a DSMS operator operates on streams and the output is another stream (selected, joined, projected, etc.). Incremental evaluation exploits the continuity of queries over time and avoids re-computation on historical data. ARGUS implements incremental evaluation methods for various stream operators. At play are factors of 100X or more in efficiency, making the difference between practical systems and unscalable concept demonstrations.

Multiple query optimization (MQO) becomes more compelling for DSMSs to effectively support long-lived large-scale concurrent continuous queries. Due to the MQO NP-completeness and asynchronous query arrivals, MQO in DSMSs has to be addressed heuristically and incrementally by adding new queries $Q$ individually into the active query evaluation plan $R$. To perform incremental MQO (IMQO), a DSMS needs to index the existing computation descriptions of $R$, identify the common computations between $Q$ and $R$, choose the optimal sharing paths, and expand $R$ to compute the final results of $Q$. ARGUS develops a comprehensive plan computation indexing and searching scheme for doing so. Particularly, it emphasizes the identification capability, and the large-scale solution. 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 recognizes syntactically-different yet semantically-equivalent predicates with canonicalization, and subsumptions between predicates and predicate sets, and support 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 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 relation model is well supported by DBMSs. Particularly, the fast search and easy update are achieved by DBMS indexing techniques, and the compact storage is achieved by following the database design methodologies. We focus on select-join-project queries in this paper. Aggregate [19] and set operations are also implemented, but will not be discussed here.

Further, ARGUS develops two sharing strategies to select the local optimal sharing paths, match-plan and sharing-selection. ARGUS also incorporates several query optimization techniques including join order optimization, conditional materialization, minimal column projection, and transitivity inference. Incorporating these techniques into the IMQO setting presents more challenges, nevertheless provides significant performance improvement over various types of queries.

ARGUS implements these capabilities atop exiting DBMS systems, such as ORACLE, for immediate practical utility and to avoid need of replicating standard DBMS functionality. Note that majority of the ARGUS work, particularly, the indexing scheme and the sharing strategies, are independent to the underlying execution engine, and are applicable to any plan-based DSMS engines as well.

In this paper, we describe the ARGUS architecture. Section 2 discusses the related work. Section 3 presents two query examples to illustrate the desirable sharing and optimization. Section 4 overviews the ARGUS architecture and incremental evaluation methods, and describes the query network structures. Section 5 describes the query network generator architecture, and the incremental sharing procedure. Section 6 details the indexing scheme and the related searching algorithms. Section 7 presents the two sharing strategies implemented in ARGUS, match-plan and sharing-selection. Section 8 presents the evaluation results. Finally, Section 9 concludes with the future work.

# 2 Related Work

The related work are presented in three database areas, DSMS, MQO, and view-based query optimization.

## 2.1 Data Stream Management Systems (DSMS)

Researchers have developed several DSMS prototypes, including STREAM, Aurora, TelegraphCQ, and NiagaraCQ. These prototypes do not address IMQO to an enough extent, particularly for indexing and searching the common computations. Also, built from scratch or from DBMS code bases, the prototypes are not immediately deployable to existing DBMS applications.

Stanford STREAM [23, 5] is a general-purpose DSMS prototype comprised of the plan generator and the execution engine. It focuses on the engine development, has rich supports on adaptive query processing [6], and implements resource sharing strategies inside the engine [3]. STREAM does not have the module to identify and share common computations among multiple queries, nor can it handle self-joins on a stream (a common operation). The only available sharing is through referencing the same output streams of previously defined queries, like view references in DBMSs.

Aurora/Borealis [2, 1], a general-purpose DSMS developed by Brandeis, Brown, and MIT, provides rich supports on distributive processing and tolerance of failures. It supports a procedural language to specify queries. With this procedural approach, much of the query sharing and optimization work is pushed to the user side, and thus the system does not have an individual module for it. However, this approach is not suitable for the large-scale query applications or the applications where the users are not expected to have extensive knowledge on the internal system.

Berkeley TelegraphCQ [8] is a general-purpose DSMS implementing a very different architecture. The plan is a set of operators built around Eddies [4], the integrated adaptive query processors. Eddies route the stream tuples through various operators to compute the final query results. TelegraphCQ [22] supports IMQO by grouping and indexing individual predicates. [21] describes the strategies to avoid too-much sharing which produces unnecessary intermediate results. Applying only shallow syntactic analysis, the indexing scheme does not identify complex common expressions, and thus misses the opportunity to perform many computation sharing operations.

NiagaraCQ [9, 10] is designed to handle large-scale queries and supports IMQO. Simple selection predicates are grouped by their expression signatures and evaluated in chains. Equi-join predicates can also be shared. However, it applies only shallow syntactic analysis and only indexing computations at predicate level. This simplified approach also confines the sharable plan structures to single predicate materialization, and restricts the applicable sharing strategies (see Section 7). NiagaraCQ applies a match-plan strategy. In contrast, ARGUS indexes computation information on the levels of literal predicates, OR predicates, predicate sets, and topology structures, and implements two sharing strategies, match-plan and sharing-selection.

Gigascope [11] is a special-purpose DSMS designed for network analysis applications. Nile [16] is a DSMS prototype on distributed sensor network applications. And CAPE [30] is a DSMS prototype concerning dynamic query re-optimization. They do not support IMQO.

Other closely related work on IMQO are Gator in Ariel [17] and Trigger Grouping in WebCQ [28].

Both implemented the chained sharing on simple selection predicates, similar to NiagaraCQ.

## 2.2 Multiple Query Optimization and View-based Query Optimization

Our IMQO approach, particularly the indexing scheme and the sharing strategies, is also closely related to but substantially different from the previous approaches targeted to MQO and view-based query optimization (VQO).[1] We first discuss the related work to the indexing scheme, then we discuss the related work to the sharing strategies.

Since MQO and VQO target to different scenarios, their indexing schemes do not suit for IMQO in DSMSs. MQO focuses on one-shot optimization where queries are assumed to be available all at a time. Therefore, the plan structures do not need to be indexed, since they are constructed from scratch and are not needed for future search and updates. In VQO, the plan structures for views also do not need to be indexed, since they are predetermined and the unmaterialized internal results are not sharable.

In most of these approaches, the common subexpressions are identified by constructing and matching query graphs [13, 7, 29]. A query graph is a DAG presenting selection/join/projection computations for a query. The matching has to be performed one query by the other. Signature (table/column references in predicates) can be hashed for early detection of irrelevant query graphs. However, many, with the same signature but irrelevant, remain and have to be filtered out by the regular semantic matching. Since it does not index any plan structure information, it can not be used for IMQO plan indexing.

In the remaining approaches, particularly for the view-matching problem in VQO, a top-down rule-based filtering method equipped with view-definition indexing is applied [14, 12]. It identifies the sharable views by filtering out irrelevant ones as soon as possible. The view-definition indexing is different from our indexing scheme in two ways. It does not index the internal plan structures for view evaluations, since the internal intermediate results of a view are not materialized and thus not sharable. It also does not support fast updates since the index is not expected to change frequently over time. In contrast, a shared continuous query plan contains materialized intermediate results to support the continuous evaluation of state operators, such as joins, and is expected to change upon new query registrations.

A sharing strategy specifies the procedure of searching the optimal sharing path in a given search space. In the MQO setting, [27] presents a global search strategy with the A*-search algorithm, and [26] uses heuristics to reduce the global search space. Their approaches assume that the queries are all available at the time of optimization, and are not applicable to the IMQO setting. NiagaraCQ implements an IMQO sharing strategy, match-plan. Match-plan matches a plan optimized for the single new query with the existing shared query plan from bottom-up. This strategy may fail to identify certain sharable computations by fixing the sharing path to the pre-optimized plan. ARGUS implements two sharing strategies, one is match-plan, and the other is sharing-selection. Sharing-Selection identifies sharable nodes on 2-way joins and locally chooses the optimal one. Actually, match-plan can be viewed as a special case of sharing-selection by always choosing the join at the lowest level among all sharable 2-way joins.

[1] View-based query optimization is a query optimization approach that identifies and uses sharable materialized views to speed up the query evaluation.

# 3 Query Examples

We present two query examples to illustrate the desirable sharing and optimization.

Consider a query $Q_1$ on big money transfers for financial fraud detections. The query links big suspicious money transactions of type 1000, and generates an alarm whenever the receiver of a large transaction (over $1,000,000) transfers at least half of the money further within 20 days using an intermediate bank. The query can be formulated as a 3-way self-join over the transaction stream $F$.

$$\begin{array}{lll}
\text{SELECT} & r1.tranid,\ r2.tranid,\ r3.tranid & \\
\text{FROM} & F\ r1,\ F\ r2,\ F\ r3 & \\
\text{WHERE} & r2.type\_code = 1000 & \\
AND & r3.type\_code = 1000 & \\
AND & r1.type\_code = 1000 & \\
AND & r1.amount > 1000000 & * \\
AND & r1.rbank\_aba = r2.sbank\_aba & \\
AND & r1.benef\_account = r2.orig\_account & \\
AND & r2.amount > 0.5 * r1.amount & * \\
AND & r1.tran\_date <= r2.tran\_date & \\
AND & r2.tran\_date <= r1.tran\_date + 20 & \\
AND & r2.rbank\_aba = r3.sbank\_aba & \\
AND & r2.benef\_account = r3.orig\_account & \\
AND & r2.amount = r3.amount & * \\
AND & r2.tran\_date <= r3.tran\_date & \\
AND & r3.tran\_date <= r2.tran\_date + 20; &
\end{array}$$

We add two predicates that can be inferred automatically by a transitivity inference module [20] from the three star-marked predicates. They are $r2.amount > 500000$ and $r3.amount > 500000$. If the inferred predicates are selective, the performance could be significantly improved.

We classify the predicates into predicate sets (PredSets) based on the table references:

$$P_1 \begin{array}{llll} r1.type\_code & = & 1000 & AND \\ r1.amount & > & 1000000 & \end{array}$$

$$P_2 \begin{array}{llll} r2.type\_code & = & 1000 & AND \\ r2.amount & > & 500000 & \end{array}$$

$$P_3 \begin{array}{llll} r3.type\_code & = & 1000 & AND \\ r3.amount & > & 500000 & \end{array}$$

$$\begin{array}{lllll}
 & r1.rbank\_aba & = & r2.sbank\_aba & AND \\
 & r1.benef\_account & = & r2.orig\_account & AND \\
P_4 & r2.amount & > & 0.5 * r1.amount & AND \\
 & r1.tran\_date & <= & r2.tran\_date & AND \\
 & r2.tran\_date & <= & r1.tran\_date + 20 &
\end{array}$$

$$\begin{array}{lllll}
 & r2.rbank\_aba & = & r3.sbank\_aba & AND \\
 & r2.benef\_account & = & r3.orig\_account & AND \\
P_5 & r2.amount & = & r3.amount & AND \\
 & r2.tran\_date & <= & r3.tran\_date & AND \\
 & r3.tran\_date & <= & r2.tran\_date + 20 &
\end{array}$$

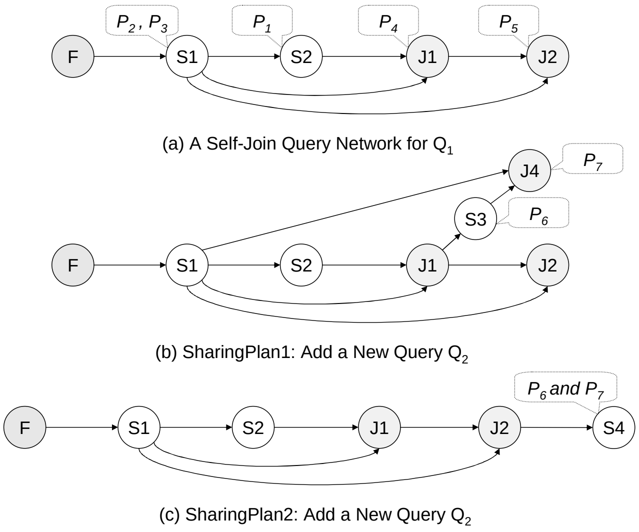
Figure 1(a) shows an evaluation plan for this query. A query evaluation plan, also called query network, is a directed acyclic graph (DAG). A node $N$ presents a set of results [2] that are obtained by evaluating a PredSet on $N$'s parent node(s). Assume the selection PredSets ($P_1$, $P_2$, $P_3$) are very selective, thus the optimal network should evaluate them first. Because PredSets $P_2$ and $P_3$ are equivalent, they share the same node $S1$. Because $P_2$ and $P_3$ subsume $P_1$ (the result set of $P_1$ is always a subset of that of $P_2$ or $P_3$, or say $P_1 \rightarrow P_2$, and $P_1 \rightarrow P_3$), thus $P_1$ can be better evaluated from node $S1$ instead of from the source node $F$ since less data needs to be processed. Assume $P_4$ and $P_5$ are equally selective. Because the input size to $P_4$ is less than that of $P_5$, $P_4$ is evaluated first for $J1$.

Query network nodes are associated with PredSets, as shown in Figure 1. Actually, there are multiple associations depending on which ancestors the PredSets are formulated on. Figure 1 shows the PredSets formulated on the very original source stream $F$ for illustration. They are different from the PredSets that are actually used to evaluate on the direct parent node(s) to obtain the results. For example, instead of $P_1$, the PredSet used to evaluate on node $S1$ to obtain $S2$ contains just one predicate $r1.amount > 1000000$, and is formulated with regard to $S1$ column names. In terms of common computation search, we need to know the very original associations. In terms of constructing the plan executable code, we need to know the direct parent associations. Thus multiple associations are indexed.

Consider a second query $Q_2$ which is the same to $Q_1$ except the time span is 10 days instead of 20 days. Thus PredSets $P_4$ and $P_5$ are changed to $P_6$ and $P_7$, respectively.

$$\begin{array}{lllll}
 & r1.rbank\_aba & = & r2.sbank\_aba & AND \\
 & r1.benef\_account & = & r2.orig\_account & AND \\
P_6 & r2.amount & > & 0.5 * r1.amount & AND \\
 & r1.tran\_date & <= & r2.tran\_date & AND \\
 & r2.tran\_date & <= & r1.tran\_date + 10 &
\end{array}$$

[2] As shown in Section 4, $N$'s results actually are comprised of two parts.



(a) A Self-Join Query Network for $Q_1$

(b) SharingPlan1: Add a New Query $Q_2$

(c) SharingPlan2: Add a New Query $Q_2$

Figure 1: Sharing Query Networks

$$P_7 \begin{array}{llll} r2.rbank\_aba & = & r3.sbank\_aba & AND \\ r2.benef\_account & = & r3.orig\_account & AND \\ r2.amount & = & r3.amount & AND \\ r2.tran\_date & <= & r3.tran\_date & AND \\ r3.tran\_date & <= & r2.tran\_date + 10 & \end{array}$$

Since $P_6$ is subsumed by $P_4$, $P_6$ can be evaluated from $J1$ as results of a selection predicate ($r2.tran\_date$ $<= r1.tran\_date + 10$) to obtain $S3$ as shown in Figure 1(b). Then $P_7$ is evaluated to obtain final results in $J4$. A better sharing choice SharingPlan2 is shown in Figure 1(c) that avoids creating the join node. Recognizing that $J2$ provides the superset of $Q_2$'s final results, $P_6$ and $P_7$ are actually evaluated from $J2$ as a selection PredSet $P_8$ to obtain $S4$, which is equivalent to $J4$ in Figure 1(b):

$$P_8 \begin{array}{llll} r2.tran\_date & <= & r1.tran\_date + 10 & AND \\ r3.tran\_date & <= & r2.tran\_date + 10 & \end{array}$$

The nodes of the shared plan should not contain all columns from their parent nodes, but only the columns in the results and the columns that are used for further evaluations. This is called *minimal column projection*.

To obtain the shared plans in Figure 1, the system needs to perform transitivity inference to infer hidden predicates, generate plans for individual queries, identify common computations and sharing paths, such as SharingPlan2, perform minimal column projection, and construct and extend the shared plan. The common computations involve predicate and PredSets that are equivalent and/or have subsumption relationship, and the associations between nodes and their PredSets. The remaining of this paper shows how ARGUS performs these tasks to construct the shared plans.

# 4 System Overview

An ARGUS continuous query is specified in SQL. *Sliding windows*, a feature commonly used in continuous queries, can be expressed by range predicates on timestamp attributes. ARGUS assumes the query conditions in where-clauses are expressed in a conjunctive normal form (CNF).

ARGUS is comprised of two components, Query Network Generator (NetGen) and Execution Engine (Engine), shown in Figure 2. Upon receiving the request of registering a new continuous query $Q$, NetGen parses $Q$, searches and chooses the sharable computations between $Q$ and the existing query network $R$, constructs a shared optimal query evaluation plan, expands the query network to instantiate the plan, records the network changes in the system catalog, and sends the updated execution code of the query network to the engine. The engine runs the execution code, and produces new results if newly arrived stream tuples match the queries. The execution is scheduled periodically, but can also be invoked upon arrivals of new data.

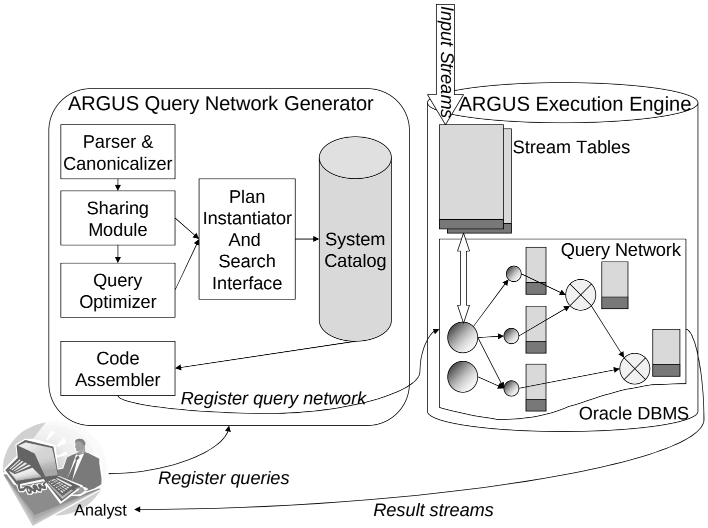


Figure 2: Using ARGUS. An analyst registers a query $Q$ with ARGUS. Query Network Generator processes $Q$, generates the query network and its execution code, records the information in the System Catalog. Execution Engine executes the code to monitor the input streams.

The engine is the underlying DBMS query execution engine. We use its primitive supports for relation operators, but not use 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 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.

In the remaining of this section, we describe the query network structures, how it incorporates incremental evaluation methods, and how it is assembled to the code that can be executed by the DBMS engine. We describe how to generate the query network in the next section. We will not discuss how the DBMS generates and executes the physical plans.

## 4.1 Query Network Structure

A query network is a directed acyclic graph (DAG). Figure 3 shows an example, which evaluates four queries. The upper part evaluates queries $Q_1$ and $Q_2$ described in Section 3, 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 intermediate or final results.

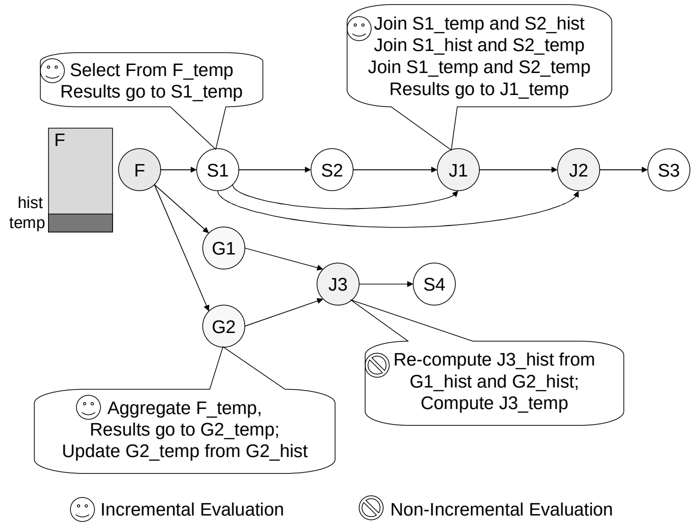


Figure 3: Execution of a shared query network. Each node has a historical (hist) table and a temporary (temp) table, here only those of node $F$ are shown. The callouts show the computations performed to obtain the new results, to be stored in the nodes temporary table. $S$ nodes are selection nodes, $J$ 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 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. 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, or the direct parent association as discussed in Section 3, presents the evaluation of a set of operators on the parent node(s) to obtain the results stored in the child node. The operator set could be a set of selection predicates (selection PredSet), a set of join predicates (join PredSet), a set of GROUPBY expressions, or a set operator (UNION, set difference, etc.). Table 1 shows the types of the operator sets and their result nodes [3].

| Operator Set | # of parents | Result Node Type |
|---|---|---|
| Selection PredSet | 1 | Selection node |
| Join PredSet | 2 | Join node |
| GROUPBY Expressions | 1 | Aggregate node |
| Set Operator | $\geq 2$ | Set operator node |

Table 1: Operator sets and result nodes

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. For example, the new results of a selection PredSet can be evaluated solely from the parent's temporary table, e.g. in Figure 3, $S1\_temp$ can be obtained by selecting from $F\_temp$.

For another example, the new results of a 2-way join PredSet can be evaluated by three small joins from the parents' historical and temporary tables, e.g. in Figure 3, $J1\_temp$ can be obtained by three joins, $S1\_temp \bowtie S2\_hist$, $S1\_hist \bowtie S2\_temp$, and $S1\_temp \bowtie S2\_temp$. Performing the three small joins is much faster than performing a large join on the whole data sets, $(S1\_hist + S1\_temp) \bowtie (S2\_hist + S2\_temp)$, since the temporary tables are much smaller than the historical tables, $|S1\_temp| \ll |S1\_hist|$, and $|S2\_temp| \ll |S2\_hist|$ [20].

For the last example, an aggregate function SUM can be evaluated by adding the new tuple values to the accumulated old aggregate values instead of revisiting the entire parent historical table, e.g. in Figure 3, $G2\_temp$ can be obtained by aggregating $F\_temp$ tuples and then adding the old aggregate values from $G2\_hist$ to the aggregate values [19].

Some operator sets can not be incrementally evaluated. Examples include holistic aggregates, such as quantiles [15], and operator sets operating on non-incrementally-evaluated nodes. In the current implementation, we do not perform incremental evaluation on post-aggregate operators, e.g. the join node $J3$ in Figure 3. Because the aggregate nodes' historical tables also need updates, and the system currently does not trace such updates, the incremental evaluations from aggregate nodes will not produce the correct results. If the system is extended to support the tracing, the incremental evaluation methods can be modified to evaluate post-aggregate nodes as well.

[3] The system only supports 2-way joins now. We plan to extend to support multi-way joins.

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 executions 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 is 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 store and compile the execution code. Then the execution code is scheduled periodical executions to produce new results.

## 4.2 Code Assembling

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, which is called the Minimal Partial Order (MPO) requirement. Any order that satisfies the MPO requirement is called a Minimal Partial Order (MPO).

One way to get a MPO list is to traverse the entire network starting from the original stream nodes. However, since the query network is recorded as a set of node entries in the system catalog relations, not in linked data structures[4], the traversal entails many system catalog accesses and is not efficient. The traversal algorithm itself is complicated since it needs to support various traversal strategies, e.g. breadth-first and depth-first, to allow flexible scheduling, which will be implemented in future.

Another way to get a MPO list is to retrieve and sort all node entries with one block system catalog access as long as each node is associated with a sort ID whose order renders a MPO. On the other hand, any one-dimensional linear MPO sort ID assignment confines to one restrict unchangeable order, which will not allow dynamic rescheduling, a useful adaptive processing technique that we plan to support in future. Such assignment is also hard to maintain when the query network expands, since adding a new node may entail the assignment update for a significant portion of nodes in the query network.

To address such problems, 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 local order within sub-network graphs. 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

[4] The reason for doing so is to provide the fast searching and easy updating to the system catalog.

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.

**Theorem 1** *Any connected sub-network graph of the query network whose nodes have the same JoinLevel must be a tree.*

**Proof:** First, we prove that any non-source node in the sub-graph has a single parent. The non-source nodes are with regard to the sub-graph, not to the entire query network.

Assume there is one non-source node $N$ that has more than one parents, then at least one of its parents is also in the sub-graph since $N$ is a non-source node. Assume the parent is $M$, then $M$'s JoinLevel must be less than $N$'s, which contradict to the sub-graph definition.

Second, we prove that there is only one source node in the sub-graph.

Assume there are more than one source nodes in the sub-graph, and two of them are $N1$ and $N2$. Since the sub-graph is connected, and $N1$ and $N2$ are source nodes (nodes without incoming edges), so they must be connected by at least one common descendant $N$ in the sub-graph. Then $N$ has more than one parent, which is impossible according to above proof.

Now, the sub-graph is a DAG with a single source node, and each non-source node has a single parent, so the sub-graph is a tree. ■

The tree is called a **local tree**. SequenceIDs are defined within local trees. The root node's SequenceID is 0, and a child node's SequenceID is always bigger than its parent's SequenceID.

When a new node $N$ is created as a leaf node of the tree, its SequenceID is assigned as $k$ plus its parent's SequenceID. In the system, we set $k = 1000$. 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 MPO order can be obtained by sorting on the JoinLevels and then on the SequenceIDs. Since multiple nodes may have the same JoinLevel and SequenceID, there are ties. Different tie resolution strategies render different MPO orders. In future, we want to apply additional information (locality) to choose optimal MPOs or rearrange MPOs for dynamic rescheduling. Such techniques may improve performance significantly when disk page swapping is inevitable and the data characteristics are changing dramatically. It is noticeable that the two-dimensional assignment is still stricter than the MPO requirement. For example, a depth-first traversal is a MPO, but violates the two-dimensional sorting criteria. Studying such legal violations may lead to finer MPO searching.

# 5 Query Network Generator

We describe ARGUS Query Network Generator in this section. Details on some important problems, particularly on the indexing scheme and the sharing strategies, are described in the following two sections.

ARGUS Query Network Generator (NetGen) generates and updates the shared query network. Given a new query $Q$, it constructs an optimal shared evaluation plan, expands the existing query network with the plan, generates the updated executable code, and register it with the engine. Figure 4 shows the architecture of NetGen. In the NetGen, the ARGUS manager, a master program, coordinates various modules to complete the query processing procedure. The procedure accesses the system catalog to lookup and update the query network information.

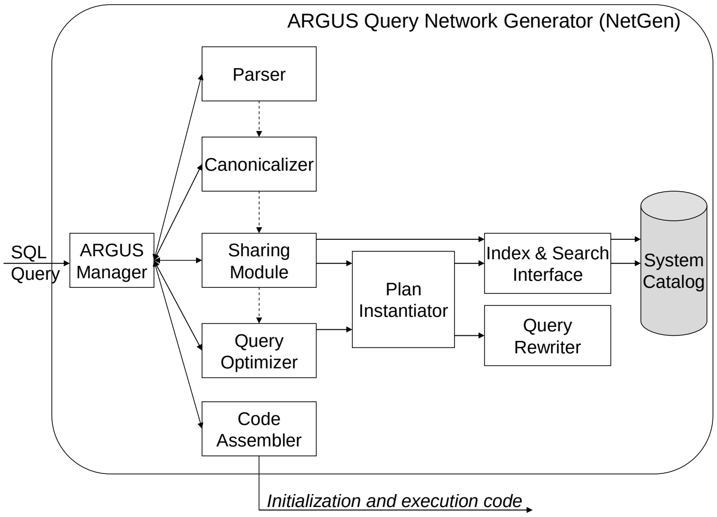


Figure 4: Query Network Generator.

ARGUS indexes all query network related information in the system catalog, a set of DBMS relations. The indexing scheme involves computation and topology indexing at four layers. First, we index literal predicates. Second, we index OR predicates (disjunction of literal predicates). Third, we index PredSets (conjunction of OR predicates). Fourth, we index the associations between the PredSets and the nodes, and the topological connections between nodes.

## 5.1 Searching Algorithms

Several algorithms are needed to assemble the retrieved information to formulate the conceptual sharable computations. Details are described in Section 6. Here we overview the functionalities.

Given a PredSet $P_{Qi}$ in the new query $Q$, the searching goal is to find a sharable node $N$ whose PredSet $P_N$ subsumes $P_{Qi}$, so that $P_{Qi}$ is either computed by $N$, or can be computed from $N$. We start the search from $P_{Qi}$'s literal predicates. For each literal predicate $\rho_{Qijk}$ in an OR predicate $p_{Qij}$ of $P_{Qi}$, we retrieve a set of sharable literal predicates $\{\rho_{RQijk}\}$. Then an algorithm checks the relationship between $p_{Qij}$ and the system-catalog-indexed OR predicates that contain any of $\{\rho_{RQijk}\}$. Once the sharable OR predicates $\{p_{RQij}\}$ are determined for each OR predicate $p_{Qij}$ in $P_{Qi}$, a similar algorithm is called to find the sharable system-catalog-indexed PredSets $\{P_{RQi}\}$ for $P_{Qi}$. Then we check the very original associations of $\{P_{RQi}\}$ to identify the sharable nodes $\{N\}$. The optimal $N$ will be chosen by a sharing strategy based on estimated costs.

For aggregate functions, the sharable nodes can be identified by finding the nodes whose GROUPBY expression set is a superset of that of the query [19]. For set operator nodes, the sharable nodes can be identified by finding the nodes whose parent set is a subset of the query's parent set.

## 5.2 Canonicalization

A literal predicate can be expressed in different ways. For example, $t1.a < t2.b$ can also be expressed as $t2.b > t1.a$. A simple string match can not identify such equivalence. To be able to match such equivalent predicates, we introduce a canonicalization procedure. It converts the syntactically-different yet semantically-equivalent literal predicates into the same pre-defined canonical form. The literal predicates are indexed in the canonical forms. Then given a new query $Q$, its canonicalized literal predicates can be matched with the system-catalog-indexed canonical literal predicates by the exact string match. Details on canonicalization are described in Section 6.

## 5.3 Column Projection

To save the disk space and improve the evaluation performance, only necessary columns should be projected into node tables. This is called Minimal Column Projection (MCP). The necessary columns include the columns appearing in the final results and the columns needed for further evaluations. The columns for further evaluations can be identified by looking at the column references in the not-yet-evaluated predicates and expressions in the where-clause, the groupby-clause, and the having-clause.

The realization of MCP becomes much more complicated when sharing is involved. When a sharable node $N$ is identified and chosen, it may not contain all the necessary columns for the new query. We need to add the missing columns to $N$. This is called *projection enrichment*. An intricate problem is that $N$'s parent(s), denoted as a node set $\{M\}$, may not contain all the missing columns either. So we also need to add the missing columns to $\{M\}$, then to $\{M\}$'s parents, and so on, until all the added columns to the nodes can be projected from their parents. This branched back-tracing process is called *chained projection enrichment*. It is implemented in ARGUS to support both sharing and MCP.

## 5.4 System Catalog

The system catalog is a set of DBMS relations that record the query network information. There are a set of algorithms to retrieve and update the relations through SQL queries, including the above mentioned sharable computation identification algorithms. The relations are classified into three categories: query network storage, coding storage, query storage, shown in Figure 5.

Query network storage is the biggest and the central category. Design details are described in Section 6. This part indexes computations including canonicalized literal predicates, OR predicates, PredSets, and GROUPBY expressions (*computation relations*); it indexes column projection information (*projection relations*); and it indexes the associations between the computations and the nodes, and the topological connections between the nodes (*topology relations*).

**Query Network Storage**

*Computation Relations:*
PredIndex
PSetIndex
GroupExprIndex
GroupExprSetIndex

*Projection Relations:*
JoinColumnMap
SelectionColumnMap
GroupColumnMap
UnionColumnMap

*Topology Relations:*
SelectionTopolgy
JoinTopology
GroupTopology
UnionTopology

**Coding Storage**
LinearNodeTable

**Query Storage**
QueryTable

Figure 5: System Catalog.

Coding storage contains one relation, LinearNodeTable, for code assembler to construct the executable stored procedures. It stores for each node the initialization and execution code blocks, the two-dimensional sort ID, the Boolean flag, and the parent Boolean flag(s). The code assembler needs to know the Boolean flags, so that in the corresponding stored procedures, they can be declared and initialized, or can be passed in as arguments.

Query storage stores the information related to the original continuous queries including the query texts, and the result tables.

## 5.5 Sharing Strategies

A sharing strategy specifies the criteria of choosing the optimal sharing path among multiple available ones. We note that the exhaustive search for a global optimal sharing path is equivalent to MQO and is NP-complete. Therefore, a practical solution needs to apply a local greedy search strategy to incrementally construct the locally optimal sharing paths.

ARGUS implements two local sharing strategies, match-plan and sharing-selection. Details are described in Section 7. Match-plan is also implemented in NiagaraCQ. It first generates an optimal plan for the single new query, then matches the sub-plans from bottom to the existing query network. On the other hand, sharing-selection first identifies the sharable nodes and then chooses the optimal one based on cost estimates. Match-plan may fail to identify certain sharable computations by fixing the sharing path to the pre-optimized plan. Sharing-plan does not have such problem. In general, sharing-selection identifies more sharable paths than match-plan, and constructs more concise query networks which run faster. This is confirmed by the evaluation.

## 5.6 Query Optimization

We need a query optimizer. When a query or a part of the query can not be shared from the existing query network, the query optimizer generates an optimal plan for the unsharable computations. The major goal is to choose an optimal join order.

The query optimizer also decides whether or not a selection PredSet is materialized based on a threshold of its selectivity (default is 0.3). If it is, a selection node is created for the selection PredSet; if it is not, the selection PredSet is unioned with a join PredSet, and is evaluated there in the unioned PredSet to obtain the join node. A selection PredSet should be materialized before a join if it is highly selective, since it significantly reduces the amount of data the join needs to work on; the selection PredSet should not be materialized if it is not selective, since it does not reduce much of the join data, but occupies more disk space, and consumes more I/O time on the materialization. The overhead may even exceed the benefit of reducing the amount of data. Confirmed by the evaluation [20], the conditional materialization provides finer-tuning toward constructing the optimal query networks.

Transitivity inference is another optimization technique implemented in ARGUS. It is applied after the query parsing. Transitivity inference explores the transitivity property of comparison operators, such as $>$, $<$, and $=$, to infer hidden selection predicates. For example, in the example in Section 3, from three existing predicates, $r1.amount > 1000000$, $r2.amount > r1.amount * 0.5$, and $r3.amount = r2.amount$, the system can infer two predicates, $r2.amount > 500000$, and $r3.amount > 500000$. Confirmed by the evaluation [20], if the inferred predicates are very selective, they improve the performance significantly by reducing the amount of the join data.

## 5.7 Query Registering

We summarize the work flow of registering a new query $Q$, given the existing query network $R$.

1. The parser parses the query to a parse tree.

2. The transitivity inference module infers the hidden predicates and adds them to the parse tree.

3. The canonicalizer canonicalizes predicates.

4. The predicate converter groups the predicates into PredSets.

5. The sharing module performs the sharing. It (a) searches the common computations between the parse tree and the query network $R$; (b) constructs a sharing plan which describes the optimal sharing path; (c) calls the plan instantiator to instantiate the sharing plan; (d) and rewrites the query parse tree to reference the shared node. Then the sharing module works on the rewritten parse tree until no more sharing can be performed.

6. The query optimizer generates an optimized plan for the remaining unsharable parse tree, calls the plan instantiator to instantiate the plan.

7. The plan instantiator indexes the computations, topologies, and columns, performs the chained projection enrichment, generates code blocks, and creates nodes, as so requested by the sharing plan or the optimized plan.

8. The code assembler assembles the code blocks into executable stored procedures, and registers them with the execution engine.

# 6 Indexing and Searching

We describe the computation indexing scheme and the related searching algorithms in this section.

The computations of a query network is a 4-layer hierarchy. From top to bottom 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 forms. And the top layer presents network topological connections.

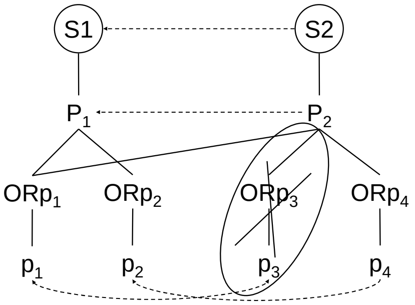


Figure 6: Computation hierarchy.

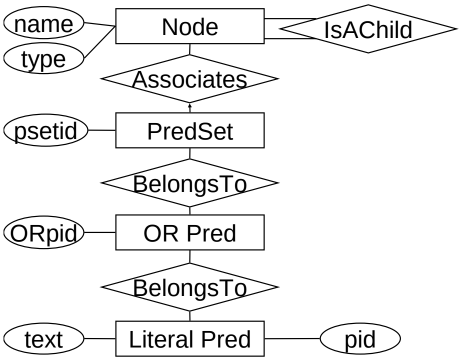


Figure 7: Hierarchy ER model.

Figure 6 shows the hierarchy for the two nodes $S1$ and $S2$ in Figure 1. The ORPreds are trivial in this example. But in general, the ORPred layer is necessary to support full predicate semantics. For the equivalent PredSets $P_2$ and $P_3$, only $P_2$ is shown. For the equivalent predicates $p_1$ and $p_3$, only $p_1$ is shown, while $p_3$ is crossed out and dropped from the hierarchy. The dashed arrows between PredSets and literal predicates indicate subsumptions at these two layers. And the dashed arrow between nodes $S1$ and $S2$ indicates the direct topology connection between them.

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 hierarchy can be presented in an ER model, as shown in Figure 7.

The reason that we do not use the linked data structure to record the query network is due to its search and update inefficiency in 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 ER model 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. We discuss these issues and their solutions in the remaining of this section. The solutions are then implemented in the final relational model.

## 6.1 Rich Syntax and Canonicalization

A literal predicate can be expressed in different ways. For example, $t1.a < t2.b$ can also be expressed as $t2.b > t1.a$. A simple string match can not identify such equivalence. For doing so, we introduce a canonicalization procedure. It transforms the syntactically-different yet semantically-equivalent literal predicates into the same pre-defined canonical form. Then the equivalence can be detected by the exact string match.

There is an 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. Therefore, we apply a triple-string canonical form, *(LeftExpression Operator RightExpression)*. *LeftExpression* is the left side of the canonicalized predicate and is the canonicalized expression containing all the column references, and *RightExpression* is the right side and is a constant. The subsumption identification can be formulated as a system-catalog look-up query on the triple strings.

Due to the extreme rich syntax and unknown semantics, e.g. user-defined functions, a complete canonicalization procedure is impossible. Previous work [14, 10, 22] apply simple approaches to identify subsumptions between simple selection predicates (e.g. $t1.a > 10$ subsumes $t1.a > 5$), equivalence of equijoin predicates and literally-matched predicates. This simplification fails to identify many syntactically-different yet semantically-related predicates that are commonly seen in practice.

Our canonicalization is more general. For example, equivalence between $r1.amount > 0.5 * r1.amount$ and $2 * r1.amount > r1.amount$, and the subsumption between $r2.tran\_date <= r1.tran\_date + 20$ and $r2.tran\_date <= r1.tran\_date + 10$ can be identified by the canonicalization but not the previous approaches.

The canonicalization applies arithmetic transformations recursively to literal predicates to convert them to predefined canonical forms. The time complexity is quadratic to the length of the predicates because of sorting. But the non-linear complexity is not a problem, since the canonicalization is a one-time operation for just new queries, and the average predicate length is far less than the degree of slowing down the process noticeably. The canonicalization does not use specific knowledge such as user-defined functions which may be time-consuming.

For non-comparison predicates, such as LIKE, IN, and NULL test, there are no exchangeable left and right sides. In such cases, we only canonicalize their subexpressions. We treat range predicates BETWEEN as two conjunctive comparison predicates. For comparison predicates on char-type data, the left and right expressions are exchangeable but can not be piled into one single side. In this case, left and right sides are canonicalized separately.

The canonicalization for numeric comparison predicates is more complex. It is a recursive transformation procedure. At each recursion, it performs pull-up, flattening, sorting, and constant evaluation. It flattens and sorts commutable sub-expressions, e.g. $a+(c+b) \Rightarrow a+b+c$, and $a*(b*c) \Rightarrow a*b*c$. It pulls up $-$ over $+$, and pulls up $/$ over $*$, e.g. $a-b+c-d \Rightarrow (a+c)-(b+d)$, and $a/b*c/d \Rightarrow (a*c)/(b*d)$. It pulls up $+$ and $-$ over $*$, e.g. $a*(b+c) \Rightarrow a*b+a*c$. And when possible, it merges multiple constants into one and converts decimal and fractions to integers.

Following shows the implemented canonicalization procedure on comparison predicates whose data types allow arithmetic operations.

- If the right side is not 0, move it to the left, e.g. $t1.a + 10 < t2.a - 1 \Rightarrow t1.a + 10 - (t1.a - 1) < 0$.
- Perform expression canonicalization on the left side.
- If there is a constant divisor or a constant fraction factor, or the constant factors are not relatively prime, multiply both sides by a proper number to make the factors relatively prime, and change the operator when the number is negative, e.g. $t1.a/2 - 1 > 0 \Rightarrow t1.a - 2 > 0$.
- Move the constant term on the left side to the right side. e.g. $t1.a - t2.a + 9 < 0 \Rightarrow t1.a - t2.a < -9$.
- If the right side is a negative number, the operator is $=$, and the left side is in the form of $MinusTerm_1 - MinusTerm_2$, change signs of both sides so the right side becomes a positive number, e.g. $t1.a - t2.a = -9 \Rightarrow t2.a - t1.a = 9$.
- Sort operands of commutable operators ($+$ and $*$) on the left side in the order of alphabet, e.g. $t3.a + t2.a + t1.a \Rightarrow t1.a + t2.a + t3.a$.

Expression canonicalization is a bottom-up recursive transformation procedure performed on an arithmetic expression, as shown below.

- Gather constant terms and evaluate them, e.g. $a+2-1 \Rightarrow a+1$. Involving functions, some constants may not be evaluated, then do the canonicalization on each function argument, and left the function call as it is.
- Flattening and sorting commutable sub-expression par-se trees, e.g. $a + (c + b) \Rightarrow a + b + c$, and $a * (b * c) \Rightarrow a * b * c$.
- Pull up $-$ over $+$, and pull up $/$ over $*$, e.g. $a-b+c-d \Rightarrow (a+c)-(b+d)$, and $a/b*c/d \Rightarrow (a*c)/(b*d)$.
- Pull up $+$ and $-$ over $*$, e.g. $a * (b + c) \Rightarrow a * b + a * c$.

## 6.2 Self-Join

We need to decide how to reference tables in the canonical forms. In the easy cases where the predicate is a selection predicate or a join predicate on different tables, any column reference in its canonical form can be presented as *table.column*, e.g. $F.amount$ without ambiguity and information loss. The direct table reference is necessary for applying the fast exact-string match.

However, when a predicate is a self-join predicate, using true table names is problematic. For example, the self-join predicate $r1.benef\_account = r2.orig\_\ account$ joins two records. The specification of joining two records is clarified by different table aliases $r1$ and $r2$. 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.

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 $\rho_1$: $F.c = 1000$, and the other a self-join predicate $\rho_2$: $T1.a = T2.b$. The canonicalized $\rho_1$ references the table directly, and is not aware of the STA assignment. But when it appears in $p_1$, we must identify its STA with respect to the self-join predicate $\rho_2$. Therefore, $\rho_1$'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. Thus an ORPred STA should be indexed in the PredSet in which it appears. The STA assignment must be consistent in the three-layer hierarchy. Particularly, a PredSet chooses one STA assignment, and propagates it down to the ORPred layer and the literal layer.

A 2-way self-join PredSet[5] has two possible STA assignments. And a k-way self-join has $k!$ assignments. This means that a search algorithm may try up to $k!$ times to find a match. The factorial issue is intrinsic to self-join matching, but may be addressed heuristically. In our implementation, supporting 2-way joins, the search on a self-join PredSet stops when it identifies an equivalent one from the system catalog. If both assignments lead to identify subsuming PredSets, the one that has less results (indicating a stronger condition) is chosen. To support multi-way self-joins, STA assignments may be tried one by one until an equivalent PredSet is found, the assignments are exhausted, or a good-enough one is found based on heuristics.

## 6.3 Subsumption at Literal Layer

Subsumptions present in the literal layer, ORPred layer, and PredSet layer. If a condition $p_2$ implies $p_1$, or say $p_2 \rightarrow p_1$, then $p_1$ subsumes $p_2$. Subsumptions are important for efficient computation sharing, since evaluating from the results of subsumed conditions processes less data and is more efficient. We want to identify existing conditions that either subsume or are subsumed by the new condition. The former directly leads to sharing, while the later can be used to re-optimize the query network.

Identifying subsumptions between PredSets is NP-hard [18, 25]. The hardness originates in the presence of correlated literal predicates in an ORPred. For example, $\{t1.a < 4\ OR\ t1.a > 5\}$ subsumes PredSet $\{t1.a > 2\ AND\ t1.a < 3\}$, but there is no polynomial algorithm that can identify such subsumption in general. On the other hand, without the correlated disjunction, detecting subsumptions is easy. For example, PredSet $\{t1.a > 1\ AND\ t1.a < 4\}$ subsumes $\{t1.a > 2\ AND\ t1.a < 3\}$, but not $\{t1.a > 2\ AND\ t1.a < 5\}$. Both acceptance and rejection can be computed in the linear time. Our subsumption identification algorithms are also heuristic and linear. The heuristic is the assumption of no correlated disjunctions.

Functional dependencies held on streams may pose additional subsumptions [5]. Searching all functional dependencies is also NP-complete [24], but heuristic algorithms may exist to find interesting ones for improving the computation sharing, which will be a future research problem.

This subsection describes how subsumptions at literal layer are detected from the triple-string canonical forms. And the next subsection describes the heuristic algorithms for doing so at the ORPred layer and PredSet layer.

[5] This is also true for literal predicates and ORPreds.

For LIKE, NULL Test, and IN predicates, no subsumption but only exact matching is performed. On comparison predicates, both subsumption and equivalence are identified. In the remaining of this subsection, we look at comparison predicates.

When *LeftExpression*s are the same, the subsumption between two literals may exist. It is determined by the operators and the comparison on *RightExpression*s. For example, $\rho_1 : t1.a < 1 \rightarrow \rho_2 : t1.a < 2$, but the reverse is not true. We define a comparable relationship between pairs of operators based on the order of the right sides.

**Definition 1** *For two literal operators $\gamma_1$ and $\gamma_2$ and an order $O$, we say $(\gamma_1, \gamma_2, O)$ is a subsumable triple if following is true: for any pair of canonicalized literal predicates $\rho_1$ and $\rho_2$, assume $\rho_1$: $L(\rho_1)\gamma_1 R(\rho_1)$, and $\rho_2$: $L(\rho_2)\gamma_2 R(\rho_2)$ where $L()$ and $R()$ are the left and right parts respectively. If $L(\rho_1) = L(\rho_2)$ , and $O(R(\rho_1), R(\rho_2))$ is true (the right parts satisfy the order), then we have $\rho_1 \rightarrow \rho_2$.*

| $\gamma_1$ | $\gamma_2$ | Order $O$ | | $\gamma_1$ | $\gamma_2$ | Order $O$ |
|---|---|---|---|---|---|---|
| > | >= | E | | < | <= | E |
| = | >= | E | | = | <= | E |
| > | >= | D | | > | > | D |
| >= | >= | D | | >= | > | D |
| = | > | D | | = | >= | D |
| < | <= | I | | < | < | I |
| <= | <= | I | | <= | < | I |
| = | < | I | | = | <= | I |

Table 2: Subsumable Triples $(\gamma_1, \gamma_2, O)$. $E$ is equal, $D$ is decreasing, and $I$ is increasing.

For example, $(<, <, Increasing)$ is a subsumable triple ($\rho_1 : t1.a < 1 \rightarrow \rho_2 : t1.a < 2$, and $O(1, 2)$ is true). Table 2 shows the implemented subsumable triples. With this, look-up queries can be formulated to retrieve the indexed subsumption literals in constant time.

## 6.4 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 $P' \in R$, such that $P$ is subsumed by, subsumes, or is equivalent to $P'$.

This subsection describes the algorithm that computes the subsumptions at the middle layers (PredSet and ORPred layers). We focus on finding the ORPreds that subsume $p$, then extend the algorithm to find ORPreds that $p$ subsumes, and finally discuss the similar subsumption algorithms on PredSet layer. Equivalent is easy given the subsuming and subsumed sets are identified; it is a unique ORPred or PredSet that is in the intersection of the two sets.

We assume non-redundant ORPred presentations in both queries and the indexed hierarchy. Particularly, any literal in an ORPred does not subsume any other one in the same ORPred. For example, if $p = \{\rho_1\, OR\, \rho_2\}$ and $\rho_1 \rightarrow \rho_2$, then $p$ is redundant and can be reduced to $p = \{\rho_2\}$. Similar non-redundancy is also assumed on PredSet presentations. Our algorithms guarantee that the non-redundancy holds on the hierarchy index as long as it holds on queries.

For illustration, we assume that each ORPred has $l$ literal predicates, each literal is subsumed by $s$ indexed literals, and each literal appears in $m$ non-equivalent ORPreds. $l$ is related to typical types of queries registered into the system, and thus can be viewed as a constant parameter.

Figure 8 shows that each of the $l$ literals in $p$ is subsumed by $s$ indexed literals, and each indexed literal belongs to $m$ different ORPreds.

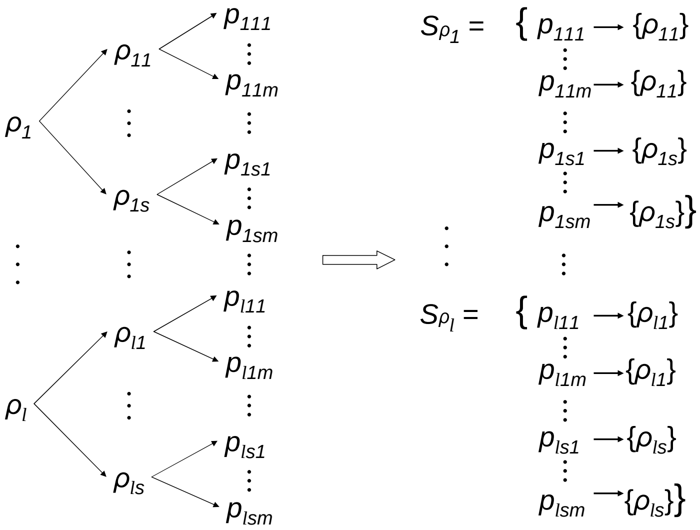


Figure 8: Subsumption and 2-level hash sets.

According to Section 6.3, given a literal $\rho_i \in p$, $i \in [1..l]$, and any pair of the $s$ literals $(\rho_{ij},\ \rho_{ih})$, $j \in [1..s]$, $h \in [1..s]$, which subsume $\rho_i$, a subsumption exists between $\rho_{ij}$ and $\rho_{ih}$, namely, either $\rho_{ij} \rightarrow \rho_{ih}$ or $\rho_{ih} \rightarrow \rho_{ij}$ is true. Along with the non-redundancy assumption, all $s{*}m$ ORPreds $\{p_{ijk} | i \in [1..s], k \in [1..m]\}$ are different to each other. Therefore $s * m \leq |R_{ORPred}|$, where $|R_{ORPred}|$ is the number of ORPreds in $R$. Generally, $s * m \ll |R_{ORPred}|$, since on average, $\rho_i$ and its related literals $\{\rho_{ij} | j \in [1..s]\}$ present narrow semantics and only appear in a small portion of $R_{ORPred}$.

Note that the ORPreds across different literal predicates, such as $p_{i_1 j_1 k_1}$ and $p_{i_2 j_2 k_2}$, $i_1 \neq i_2$, could be legitimately equivalent. In fact, we want to identify those ones and check whether they subsume $p$.

The subsumption algorithm uses a data structures called 2-level hash set. A 2-level hash set $S$ is a hashed nested set. The set elements, called top-level elements or hash keys, are hashed for constant-time accesses. The set of these elements is denoted as $keys(S)$. Each element $p \in keys(S)$ points to a bottom-level set whose elements are also hashed and is denoted as $S(p)$. Conceptually, a top-level element $p$ presents a set identifier, and its nested set $S(p)$ contains the elements that are currently detected as belonging to set $p$.

Shown in Figure 8, for each literal $\rho_i$, the subsuming literal predicates and their ORPreds form a 2-level hash set $S_{\rho_i}$. The ORPreds are the $s * m$ unique top level elements, and form the set $keys(S_{\rho_i})$.

Each ORPred points to the set of the literal predicates that belong to it. In Figure 8, the bottom-level sets are singleton sets whose elements are literals that subsume $p$.

We define a binary operation $\varUpsilon$-intersection $\cap_{\varUpsilon}$ on 2-level hash sets. The purpose is to find the sets, identified by the top-level keys, that appear in both operand sets, and to merge the currently-detected set elements in them.

**Definition 2** *Given two 2-level-hash sets $S_1$ and $S_2$, we say $S$ is the $\varUpsilon$-intersection of $S_1$ and $S_2$, denoted as $S = S_1 \cap_{\varUpsilon} S_2$, if and only if following is true: $S$ is a 2-level-hash set, $keys(S) = keys(S_1) \cap keys(S_2)$, and $\forall k \in keys(S)$, $S(k) = S_1(k) \cup S_2(k)$.*

When intersecting two 2-level hash sets $S_1$ and $S_2$, only the common top-level keys, namely, the ones appearing in both $S_1$ and $S_2$, are preserved; others, appearing in one set, but not the other, are discarded in the result set. For any preserved key $p$, its nested set is the union of $p$'s nested sets in $S_1$ and $S_2$. $\varUpsilon$-intersection can be computed in the time of $O(|keys(S)| * Average_{p \in keys(S)}|S(p)|)$ where $S$ is either $S_1$ or $S_2$. In Figure 8, the time of $\varUpsilon$-intersecting two hash sets is $O(s * m)$.

**Algorithm 1** ***Subsumed_ORPreds***
***input***: *p, R;* ***output***: *SubsumedSet(p)*

```
for each literal $\rho_i \in p$, $i \in [1..l]$
    $S_{\rho_i} := \{p_{ijk} \Rightarrow \{\rho_{ij}\} \mid \rho_{ij} \in p_{ijk},\ p_{ijk} \in R_{ORPred},$
                $\rho_i \rightarrow \rho_{ij},\ j \in [1..s],\ k \in [1..m]\};$

$I := {\cap_{\varUpsilon}}_{i=1}^{l} S_{\rho_i};$

$SubsumedSet(p) = \{\};$
for each key $p' \in keys(I)$
    if $|I(p')| = |p|$
        $SubsumedSet(p) + := p';$
```

The subsumption algorithm shown above finds all ORPreds in $R_{ORPred}$ that subsume $p$. It constructs $S_{\rho_i}$, $i \in [1..l]$, $\varUpsilon$-intersects them to $I$, and checks the satisfying ORPreds in $I$. $|I(p')|$ is the number of elements in $I(p')$, namely, the number of literals in $p'$ that subsume some literal in $p$. And $|p|$ is the number of literals in $p$. The check condition $|I(p')| = |p|$ means that if each literal in $p$ is subsumed by some literal in $p'$, then $p$ is subsumed $p'$. The time complexity is easy to be shown as $O(l * s * m)$.

The algorithm *Subsume_ORPreds* that finds all ORPreds in $R_{ORPred}$ that $p$ subsumes is very similar to *Subsumed_ORPreds*, except that the 2-level hash sets are constructed from the literals that are subsumed by $p$'s literals, and the final check condition is $|I(p')| = |p'|$, saying that if each literal in $p'$ is subsumed by some literal in $p$, then $p'$ is subsumed by $p$.

The algorithms can be easily extended to identify the subsumptions and equivalence at the PredSet layer. In that case, the top-level hash keys are the PredSet IDs and the bottom-level elements are ORPred IDs. The final check conditions dictate that a PredSet $P$ is subsumed another $P'$ if $P$ is subsumed by all literal predicates in $P'$.

Assume that each PredSet has $k$ ORPreds, each ORPred is subsumed by $t$ indexed ORPreds, and each ORPred appears in $n$ different PredSets. The time complexity of the PredSet-layer algorithm is $O(k*l*s*m+k*t*n)$ including the $k$ calls of $Subsumed\_ORPreds$. Note that $t*n \leq |R|$ given the non-redundancy assumption. $|R|$ is the number of the searchable PredSets in $R$ and is also the number of nodes in $R$. Generally, $t*n \ll |R|$ since on average the PredSets related to $p$ appear in only a small portion of the indexed PredSets. Therefore, the algorithm takes only a small portion of time $O(k*l*|R_{ORPred}|+k*|R|)$ to compute.

If the sharable PredSets are searched by matching PredSets and ORPreds one by one, the searching will take the time of $O(k^2*l*|R|)$ since $k$ new ORPreds need to match $|R|*k$ existing ORPreds and each match computes on $l$ literal predicates. Although it is also linear to $|R|$, the factor is larger and it will be much slower on large scales.

## 6.5 Topology Connections

PredSets are associated with nodes. A PredSet $P$ presents the topological connection between the associated node $N$ and $N$'s ancestors $\{A\}$. Namely, the results of $N$ are obtained by evaluating $P$ on $\{A\}$. A node $N$ is associated with multiple PredSets depending on the different 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 does not provide the fast searching. In Figure 9(a), assume a selection PredSet $P$ from stream table $B1$ can be evaluated from any of $S1$, $S2$, $S3$, $S4$, and $S5$, while $S5$ is the best one to share. If only DParent is recorded, $S5$ has to be found by a chained search process that needs to check the sharability of nodes along the way from $B1$ to $S5$. The process also needs to deal with branches, e.g. in Figure 9(a) searching the descendants of $S1$ as well.

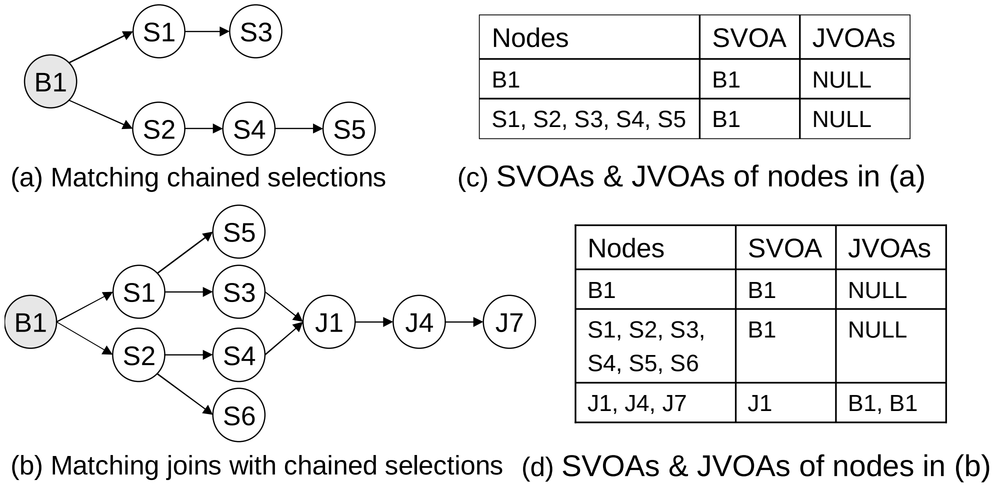
Same problems exist for join PredSets. In Figure 9(b), assume a self-join $P$ on stream table $B1$ can be evaluated from $J7$. If only DParents are recorded, $J7$ can not be found immediately. The chained search process must search all $B1$'s descendants up to the end of its next join depth. Both chained processes have the time complexity of linear to the total size of the chains.

One solution is recording all PredSets associated with a node $S$. Particularly, for any ancestor $A$ of $S$, the PredSet between them is recorded. In this approach, the number of PredSets to be recorded is higher than linear of $|R|$. Assume the average number of descendants of a node is $m$, and average number of branches is $k$, thus $km \approx |R|$. Then the number of PredSets is $O(km^2)$. Also the redundant indexing leads to redundant search.

A better choice is recording only two more PredSets for each node $N$. These PredSets are associated with nodes called $N$'s SVOA and $N$'s JVOAs. Figures 9(c) and 9(d) show the SVOAs and JVOAs for nodes in (a) and (b).

**Definition 3 (SVOA)** *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.*

**Definition 4 (JVOA)** *A join node $N$'s JVOAs are the closest ancestor nodes that are either join nodes*



(a) Matching chained selections

| Nodes | SVOA | JVOAs |
|---|---|---|
| B1 | B1 | NULL |
| S1, S2, S3, S4, S5 | B1 | NULL |

(c) SVOAs & JVOAs of nodes in (a)

(b) Matching joins with chained selections

| Nodes | SVOA | JVOAs |
|---|---|---|
| B1 | B1 | NULL |
| S1, S2, S3,<br>S4, S5, S6 | B1 | NULL |
| J1, J4, J7 | J1 | B1, B1 |

(d) SVOAs & JVOAs of nodes in (b)

Figure 9: Multiple topology connections.

*(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. Note that SVOAs and JVOAs present local topological connections within and across one join depth. The common computations identified at the local level are fed to a sharing strategy to search for optimal sharing paths. In ARGUS, we implemented two local strategies that perform sharing one join depth a time.

A semantically equivalent predicate may appear in DPredSet, JVOAPredSet, and SVOAPredSet in different forms for a given node. For example, as shown in Figure 9(b), assume a self-join predicate $p_1$ from stream base table $B1$ is actually evaluated as a selection predicate from node $J4$ to obtain $J7$. Then for node $J7$, $p_1$ will appear as the original self-join predicate from $B1$ in JVOAPredSet, as a selection predicate from $J1$ in SVOAPredSet, and as a selection predicate from $J4$ in DPredSet. Automatic conversions between these forms are needed and implemented. We also need and implement the union and difference operations on PredSets, which should ensure the non-redundancy requirement.

## 6.6 Relational Model for Indexing

Now we consider converting the indexing ER model to the relational model.[6] Two adjustments are made. First, the relations that index literal predicates and ORPreds are merged into one, *PredIndexing*, based on the assumption that ORPred are not frequent in queries. This allows a literal predicate to appear multiple times in *PredIndexing* 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

[6] The schema described in this subsection is simplified.

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.

| **PredIndex** | | | **PSetIndex** |
|---|---|---|---|
| ORPredID | Node1 | LeftExpr | ORPredID |
| LPredID | Node2 | Operator | PredSetID |
| UseSTA | STA | RightExpr | STA |

| | **SelectionTopology** | |
|---|---|---|
| Node | DirectParent | JVOA1 |
| IsDISTINCT | DPredSetID | JVOA2 |
| SVOA | SVOAPredSetID | JVOAPredSetID |

| | **JoinTopology** | |
|---|---|---|
| Node | DirectParent1 | JVOA1 |
| IsDISTINCT | DirectParent2 | JVOA2 |
| | DPredSetID | JVOAPredSetID |

Figure 10: System Catalog Schemas

Figure 10 shows the indexing relation schemas. In *PredIndexing*, ORPredID is the ORPred identifier, and LPredID is the sub-identifier of the literal within the ORPred. The combination of ORPredID and LPredID is the primary key of *PredIndex*. *Node1* and *Node2* records the ancestor tables from which the literal is evaluated. *LeftExpression*, *Operator*, and *RightExpression* are the triple-string canonical form of the literal. If the literal is a selection predicate in an self-join ORPred, *STA* is used, otherwise it is *NULL*. If the literal is a self-join predicate, or *STA* is used, the binary attribute *UsingSTA* is set, otherwise, it is *NULL*.

*PSetIndex* indexes the PredSets. The primary key is the combination of *PredSetID* and *ORPredID*, indicating which ORPred belongs to which PredSet. STA is used when the ORPred is a selection but the PredSet is a self-join.

In the topology relations, *Node* is the primary key. The binary *IsDISTINCT* indicates whether the duplicates are removed. The remaining attributes are described in Section 6.5. *JoinTopology* does not need to index SVOA.

# 7 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 a 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.

Figures 11 & 12 illustrate the difference between sharing-selection and match-plan. Assume the existing query network $R$ (Figures 11(a) & 12(a)) performs a join on table $B_1$ and $B_2$, and the results are materialized in table $J_1$. Assume the new query $Q$ performs two joins, $B_1 \bowtie B_2$ and $B_2 \bowtie B_3$, and its optimal plan (Figure 11(b)) performs $B_2 \bowtie B_3$ first . From the viewpoint of match-plan, the bottom join $B_2 \bowtie B_3$ is not available in $R$, thus no sharing is available. It expands $R$ to a new query network $R_m$ (Figure 11(c)). From sharing-selection, both of the joins (Figure 12(b)) are matched against $R$ to see whether it has been computed in $R$. In this example, $B_1 \bowtie B_2$ has, while $B_2 \bowtie B_3$ has not. Sharing the results of $J_1$ with $B_1 \bowtie B_2$, the network is expanded to $R_s$ (Figure 12(c)) which has less join nodes. In general, sharing-selection identifies more sharable paths than match-plan, and constructs more concise query networks which run faster. Actually, the match-plan method can be viewed as a special case of sharing-selection by applying a constraint: always select from bottom-level predicate sets. Match-plan and sharing-selection have the same time complexity of $O(kl)$ where $k$ is the average number of branches of a node, and $l$ is the average number of JoinLevels. $l$ is actually the typical number of joins in queries, which is a small integer, less than 15, etc.

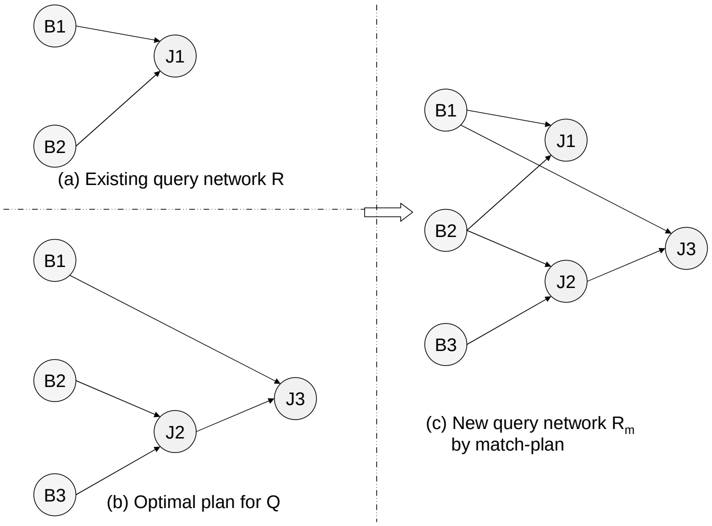
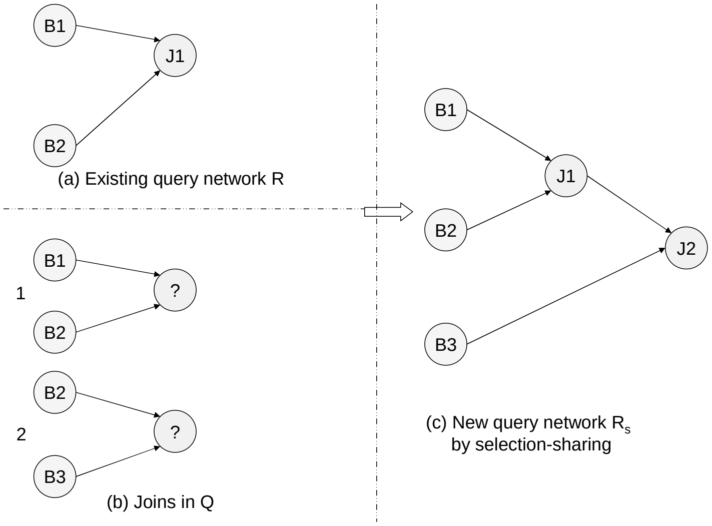


Figure 11: Match-Plan.

In both strategies, we need to choose an 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 $P$. For example, in Figure 9(a), assume $P$ can be evaluated from $S5$ to a new node. Then the cost of sharing $S5$ is the table size $S5$. Now assume $S5$ is associated with an equivalent PredSet to $P$, then no further evaluation is needed, and the cost is 0. It is possible that multiple PredSets can be shared this way (with cost 0), then future costs are used for choosing among these candidates. Future cost is still defined as the size of the sharing node. It is so called because it is the number of records to be processed from the sharing node in the next JoinLevel.

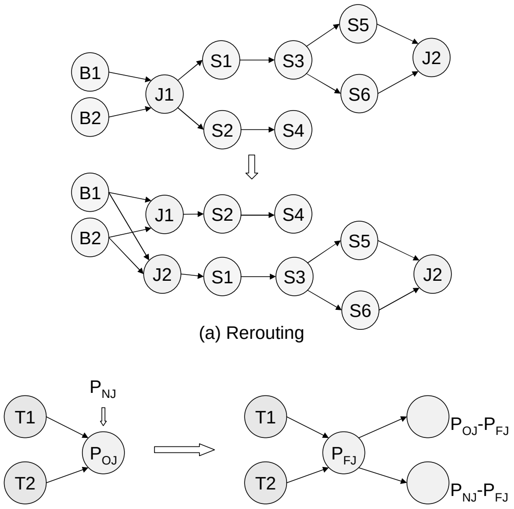


(a) Existing query network R

(b) Joins in Q

(c) New query network $R_s$ by selection-sharing

Figure 12: Sharing-Selection.



(a) Rerouting

(b) Restructure Sharing

Figure 13: Complex Sharing.

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 choose the better sharing path as shown in SharingPlan2 in Section 1. This choice is applied by both sharing-selection and match-plan.

Given the identified common computations, more complex sharing strategies, e.g. *rerouting* and *restructuring*, may be applied as well. *Rerouting* occurs after a new node is created. Once a new node is created, there may be a set of old nodes that can be evaluated from the new node. Disconnecting the old nodes from their current DParent(s) and rerouting them to be evaluated from the new one may lead to a better shared query network. Figure 13(a) shows the rerouting after a new node $J2$ is created. In this example, $S1$ can be better evaluated from $J2$, i.e. $|J2| < |J1|$, then $S1$ and its descendants are rerouted to $J2$. Restructuring is reoptimization to local topological structures when new computations are added into the network. One example is splitting computations in a join PredSet to allow multiple queries to share the same join results, as shown in Figure 13(b). The choices to perform these topological operations should be decided by cost models, and are also applicable in adaptive processing. The cost models should capture the stream distribution changes that outdate the original network, and guide the rerouting and restructuring procedures for adaptive reoptimization. The sharing strategies presented so far are local greedy optimizations bounded by join depths. Beyond, more aggressive optimization strategies can be performed by looking ahead along sharable paths, probably with heuristic pruning.

# 8 Evaluation Study

The evaluation shows the effectiveness of sharing and canonicalization, and compares the match-plan and sharing-selection. [20, 19] show the evaluation on incremental query evaluation methods and the aggregate queries.

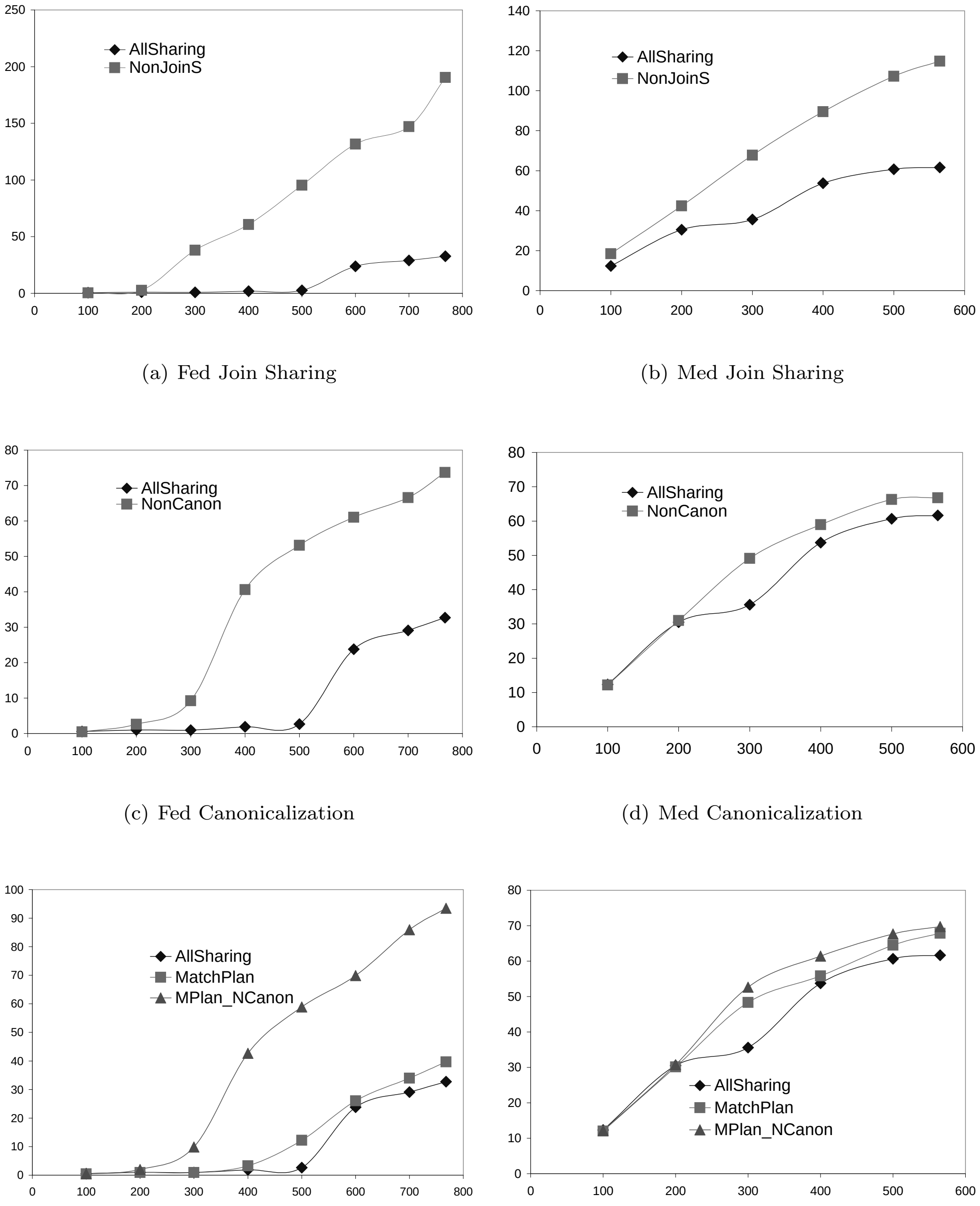
The experiments were conducted on an HP PC computer with Pentium(R) 4 CPU 3.00GHz and 1G RAM, running Windows XP.

Two databases are used. One is the synthesized FedWire money transfer transaction database (Fed) with 500000 re-cords. And the other is the Massachusetts hospital patient admission and discharge record database (Med) with 835890 records. Both databases have a single stream with timestamp attributes.

To simulate the streams, in the order of time, we take the first part (300000 records for Fed and 600000 for Med) of the data as historical data, and simulate the arrivals of new data incrementally. Query networks are evaluated on 10 incremental data sets 11 times for each set. Each incremental data set contains 4000 new records.

We use 768 queries on Fed and 565 queries on Med. These queries are generated systematically. First, interesting queries arising from applications are formulated manually as query seeds. The queries are categorized based on the n-way joins and the semantics of selection and join predicates. Changeable query constants are identified. Then more queries are generated by varying the constants of the seed queries. For example, constants such as amount in selection predicates and span of days in join predicates vary for subsumption sharing on both selection and join nodes. Section 1 presents one type of these queries.

There are 32 query categories in Fed. They vary in join conditions and the number of self-joins from 2-way up to 5-way self-joins. Results of less-join queries can be shared by more-join queries. There are 8 query categories in Med. Particularly, 3 big categories, counting for 414 queries in total, monitor multiple occurrences of various contagious diseases in local areas within a given time window. We measure



(a) Fed Join Sharing

(b) Med Join Sharing

(c) Fed Canonicalization

(d) Med Canonicalization

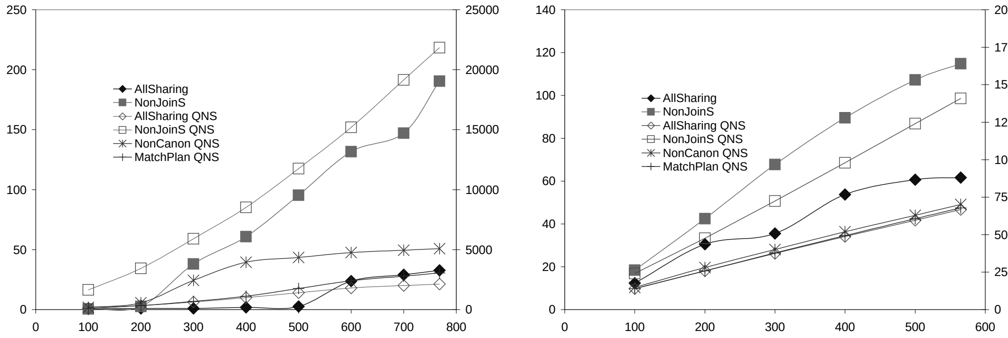
(e) Fed MatchPlan

(f) Med MatchPlan

Figure 14: Evaluation. X: number of queries; Y: total execution time in seconds.

the system performance by scaling over the number of queries, where the queries are interleaved by the categories.

We compare performance of four query network generation configurations, AllSharing, NonJoinS,



(a) Fed Weighted Network Size

(b) Med Weighted Network Size

Figure 15: Evaluation. X: number of queries; Y: total execution time in seconds; Second Y: weighted query network size.

NonCanon, and Match-Plan, as shown in Table 3. 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. Figure 14 shows the times to evaluate multiple queries scaling from 100 queries to 768 queries for Fed, and 565 queries for Med. 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 3: Network Generation Configurations. Functionality enabled: Y; disabled: N.

*Weighted query network size (QNS)* is the weighted sum of numbers of various types of nodes in the network, to roughly present the network size and its execution cost. Selection nodes have less weights than join nodes, since join is more time consuming. Lower JoinLevel nodes have larger weights, since they operate on more data tuples. Figures 15(a) and 15(b) show the QNSs of the query networks built by the different configurations. To correlate them with the actual performance, We also recap AllSharing and NonJoinS execution times in the figures. The figures indicate that the execution times are linear to the weighted size of the query networks in general. We also observe the sub-linearity when the network is very small (Figure 15(a) on Fed), we believe this is due to the fact that the underlying DBMS does a good job on buffer and cache management on relatively small data sets. We expect that optimizing the node evaluation order, e.g. grouped evaluation on local trees, will alleviate the I/O bottleneck problem for large-scale queries. Such cache-aware optimization remains a challenge for future work.

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.

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, as shown in the examples in Section 1, 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 implication relations between such join predicates. Thus the sharing leads to more significant reduction in the number of join nodes.

In Figures 14(e) and 14(f), 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.

# 9 Conclusion

Seeking practical solutions for matching highly dynamic data streams with multiple long-lived continuous queries becomes increasingly demanding. ARGUS, a stream processing system, addresses this problem by supporting incremental evaluation, query optimization, and IMQO. Particularly, ARGUS presents a comprehensive computation indexing scheme to search general sharable computations. It introduces a canonicalization procedure to index semantically-equivalent predicates. Beyond join order optimization, ARGUS implements several other optimization techniques, including conditional materialization, minimal column projection, and transitivity inference. Evaluations on each single technique, shown in this paper and previous papers [20, 19], demonstrate significant performance improvement over general or specific queries, up to well over one hundred fold speed up.

ARGUS is built atop of a DBMS to provide the value-adding stream processing functionalities to existing database applications. However, its architecture is designed in mind to support DSMS engines as well and will be integrated with DSMSs as they maturate. Further, the architecture is designed to accommodate adaptive processing techniques, such as query re-optimization, and dynamic rescheduling, which will be our future work.

# References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] A. Arasu. *Continuous Queries over Data Streams.* PhD thesis, Stanford University, 2006.

[4] R. Avnur and J. 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.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[6] S. Babu. *Adaptive Query Processing in Data Stream Management Systems.* PhD thesis, Stanford University, 2005.

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

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, January, 2003.

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

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[11] C. Cranor, T. Johnson, O. Spataschek, and V. 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.

[12] D. DeHaan, P.-Å. Larson, and J. Zhou. Stacked indexed views in Microsoft SQL Server. In *SIGMOD Conference*, pages 179–190, 2005.

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

[14] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, 2001.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[16] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[17] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, Sydney, Austrialia, March, 1999.

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

[19] C. Jin and J. Carbonell. Incremental Aggregation on Multiple Continuous Queries. In *Proc. of the 16th International Symposium on Methodologies for Intelligent Systems*, Bari, Italy, 2006.

[20] C. Jin, J. Carbonell, and P. Hayes. ARGUS: Rete + DBMS = Efficient Continuous Profile Matching on Large-Volume Data Streams. In *Proc. of the 15th International Symposium on Methodologies for Intelligent Systems*, pages 142–151, Saratoga Springs, NY, 2005.

[21] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

[22] S. Madden, M. A. Shah, J. M. Hellerstein, and V. 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.

[23] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. 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.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 3rd edition, 2003.

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

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

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

[28] W. Tang, L. Liu, and C. 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.

[29] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD Conference*, pages 105–116, 2000.

[30] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In G. Weikum, A. C. König, and S. Deßloch, editors, *SIGMOD Conference*, pages 431–442. ACM, 2004.