Optimizing Multiple Continuous Queries

Chun Jin

CMU-LTI-06-009

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

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

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
In Language and Information Technologies

This dissertation is dedicated to my dear mother, Yueying.

Abstract

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

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

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

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

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

Acknowledgements

I am extremely grateful to my Ph.D. advisor, Jaime Carbonell, for all his guidance and help he gave me. He has been a wonderful role model for me as a researcher, a teacher, and a leader. He brought me into the amazing world of research, guided me to approach and work through difficult problems, inspired me to look far beyond what we have now, helped me enhance my writing and presentation, and always encouraged me to explore challenging problems in the directions I am interested in. His continuous and generous guidance and support had made my completion of the Ph.D. possible, particularly when I went through the hard time of balancing my research and family. He helped me get the best out of both. His impact will continue on my future career and the rest of my life.

I am very grateful to my committee, Christopher Olston, Jamie Callen, and Phil Hayes. Christopher gave me lots of insightful suggestions on stream processing, inspired me to think sharply about the strength and the weakness of different approaches, and encouraged me to talk and work with other researchers. Jamie gave me wonderful suggestions on designing and analyzing experiments, encouraged me to take a broader and longer vision on research, and gave me lots of valuable advice on doing research as a career. Phil gave me lots of support as one of the project leaders, pointed out interesting problems that were not apparent at the beginning, and gave me many valuable suggestions on the system design and experiment design. I enjoyed working with them and benefited from them enormously.

I thank the following faculties, staffs, fellow students, and friends, who made my years at CMU a very memorable and beneficial experience: Anastassia Ailamaki, Santosh Ananthraman, Tom Ault, Mary Jo Bensasi, Matthew Bilotti, Alan Black, Lenore Blum, Dan Bohus, Ralf Brown, Paul Carpenter, Shimin Chen, Datong Chen, Yee Man (Betty) Cheng, Ananlada (Moss) Chotimongkol, Dwight Dietrich, Meryem Donmez, Maxine Eskenazi, Christos Faloutsos, Fang Fang, Eugene Fink, Steve Fienberg, Michele Di Pietro, Ariadna Font-Llitjos, Bob Frederking, Cenk Gazen, Aaron Goldstein, Jade Goldstein, Lingyun Gu, Benjamin Han, Peggy Heidish, Fei Huang, Yi-Fen Huang, Brooke Hyatt, Peng Jia, Qin Jin, Rong Jin, Rosemary Jones, Szu-Chen (Stan) Jou, Nancy Klancher, Jeongwoo Ko, John Kominek, Alexandros Labrinidis, John Lafferty, Chad Langley, Alon Lavie, Guy Lebanon, Lori Levin, Fan Li, Tien-ho (Henry) Lin, Fang Liu, Yan Liu, Jie Lu, Nianli Ma, Ganesh Mani, Johny Mathew, Tom Minka, Teruko Mitamura, Christian Monson, Thuy Linh Nguyen, Eric Nyberg, Paul Ogilvie, Jiazhi Ou, Jiayu (Tim) Pan, Yue Pan, Erik Peterson, Kathrin Probst, Yanjun Qi, Pradipta Ray, Radha Rao, Monica Rogati, Roni Rosenfeld, Alex Rudnicky, Norman Sadeh, Kenji Sagae, Christopher Scaffidi, Andrew Schlaikjer, Tanja Schultz, Minglong Shao, Luo Si, Li Su, Oznur Tastan, Sebastian Thrun, Stefanie Tomko, Alicia Tribble, Tiankai Tu, Alex Waibel, Haiyan Wang, Jianqun Wang, Zhirong Wang, Larry Wasserman, Adele Weitz, Daniel Wilson, Jeannette Wing, Wen Wu, Yinglian Xie, Eric Xing, Peng Xu, Wei Xu, Rong Yan, Hui (Grace) Yang, Yimin Yang, Yiming Yang, Stacey Young, Jun Yang, Danni Yu, Hua Yu, Chengxiang Zhai, Jian Zhang, Rong Zhang, Yi Zhang, Ying (Joy) Zhang, Bing Zhao, Jie Zhu, and Xiaojin Zhu.

I owe my deepest gratitude to my mother, Yueying, who has given me continuous and tremendous support she could ever since I came to this world and all the way up to now. The completion of the Ph.D. and my future contributions are probably the best presents I could ever give back to her.

Finally, I want to thank the rest of my family, my father, Hongsheng, my husband, Zhu, my daughters, Victoria and Gloria, and my brother, Jiang, for their continuous support, patience, and love. They made my life enjoyable and rewarding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

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

A traditional online transaction processing (OLTP) DBMS is designed to process multiple concurrent transactions efficiently and supports precise data storage with infrequent changes. And an online analytic processing (OLAP) DBMS is optimized to match complex analytic *ad hoc* queries on large-scale warehouse data efficiently. Differently, stream applications require continuous query matching over fast-changing data streams and produce new results continuously. The traditional DBMSs do not provide an efficient mechanism to support continuous query matching. While triggers can be defined to simulate the continuous query matching upon data changes, the method is not scalable, the triggers can not be shared, and the functionalities are limited due to the ACID and other design constraints [98]. Moreover, DBMSs do not systematically support efficient continuous matching algorithms that are vital for performance on stream processing. Many query operators can be implemented to efficiently produce new results on new tuples without or with bounded accesses to the historical data, which we call **incremental evaluation**. For example, a selection predicate can be evaluated just on the new tuples without accessing any historical data.

Another missing component is the computation sharing module. While multiple query optimization (MQO) [108, 102] has been studied since late 1980's, the techniques are not implemented in commercial DBMS since queries on historical data are typically discarded after a single search. However, since multiple concurrent continuous queries tend to be persistent, computation sharing is appropriate for stream applications and even a must for applications dealing with large-scale queries (pub/sub systems) or with intensive query dynamics (complex and evolving intelligence analysis tasks). Computation sharing can lead to hundreds-fold performance improvement. And the larger the number of concurrently active queries is, the more significant benefit is obtained.

Two problems add complexities to the implementation of a practical sharing component. First, queries arrive intermittently, not in batch, which leads to constant changes of the shared query plan. Second, since global optimization on multiple queries is known to be

NP-complete [108, 81], it is impractical to perform one-shot full optimization on large-scale queries, not to say doing it repetitively. A practical solution is to develop a system that adds new queries individually into an existing shared query evaluation plan to obtain reasonably good performance via local optimization. We call this approach **incremental multiple query optimization (IMQO)**. With this approach, the system needs to store existing query computations, identify the common computations between the new query and the existing query plan, choose optimally among multiple sharing paths, and add unsharable new computations to the plan.

This thesis addresses incremental evaluation, IMQO, and several query optimization techniques pertinent to continuous queries. The resulting techniques are implemented in a prototype called ARGUS.

1.1 Incremental multiple query optimization (IMQO)

The most distinguishable feature of ARGUS, comparing to other research prototypes on data stream management systems (DSMS), is its comprehensive IMQO framework. While computation sharing has been widely accepted as an important component of a DSMS, it is often missing or underdeveloped in existing research prototypes. For example, STREAM is a general-purpose DSMS prototype developed from scratch by Stanford University with focus on adaptive processing, extended stream query languages, and execution engine architecture. The STREAM architecture is designed to support large-scale concurrent continuous queries and executes a shared query plan that outputs multiple result streams. Algorithms realizing a range of resource sharing strategies are implemented. However, the prototype does not develop or implement the sharing module. It does not recognize the sharable computations across multiple queries, and does not support incremental addition of new queries. See Section 2.1 for details.

In another example, NiagaraCQ is a publish/subscribe prototype that matches large-scale subscriber queries with Internet content changes. It implements an incremental sharing framework similar to ARGUS. A new query Q can be incrementally added into a shared plan R by identifying sharable computations between Q and R and expanding R with new unsharable computations. However, NiagaraCQ applies a simplified approach to identify sharable computations, i.e. exact string match and shallow syntactic analysis to identify equivalent and subsumption predicates. This largely limits the potential improvement offered by computation sharing; see examples in Section 1.5. In NiagaraCQ, simple selection predicates are grouped by their expression signatures and evaluated in chains, and equi-join predicates can also be shared. But the identification is limited to the predicate level. Such limited findings restrict the construction of efficient shared plans. For example, it can not identify sharable nodes which present the results of a group of predicates (PredSet). Without topological association, such findings also limit the sharing strategies to be applied. See Sections 2.1 and 6.3.

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

- Support general query types, e.g. selection-join-projection queries, aggregate queries, set operation queries, and their combinations.
- Support general plan structures, e.g. materializing the results of grouped predicates.
- Support compact indexing storage, fast computation search, and easy index update and plan expansion for large-scale query applications.

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

- computation indexing,
- common computation identification,

- sharing path selection,
- and indexing update and plan expansion.

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

1.1.1 Indexing and Searching

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

Common subexpression identification is also known to be NP-hard [75, 99], and thus has to be addressed heuristically. Secondly, identifying sharable computations from the shared plan structures and topologies, not just from the query semantics, adds one more layer of complexity. Thirdly, to scale to a large number of queries, the scheme and algorithms should support compact index storage, fast index search and easy index update. And finally, since the shared plan dynamically evolves as new queries are registered, and is subject to local re-optimization to adapt to data distribution changes, the indexing scheme should provide enough information for fast constructing, updating, and rearranging the executable query evaluation plan.

Previous work on DSMSs, such as NiagaraCQ, STREAM, and other work described in Chapter 2, either do not develop or underdevelop the support of the common computation indexing and identification. Previous work on MQO [107, 48, 27, 31] and view-based query optimization [56, 129] address the common computation identification problem specifically, but do not concern with plan topologies, compact storage, and easy update, and use quite different approaches from this thesis; see Chapter 2.

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

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

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

1.1.2 Sharing Strategies

We present two sharing path optimization strategies for selection-join queries, match-plan and sharing-selection. Match-plan matches a plan optimized for the single new query with the existing query network with bottom-up search. This strategy fails occasionaly to identify certain sharable computations by fixing the sharing path to the pre-optimized plan.

Match-plan is also applied by NiagaraCQ [33, 34] on non-canonicalized single predicates (not on PredSets). Sharing-Selection identifies sharable nodes and chooses the optimal sharing path.

We present two sharing strategies for aggregate queries. The first one is to choose the optimal sharing node when there are multiple sharable ones. And the second is *rerouting*. After a new node B is created, the system checks if any existing nodes can be improved by being evaluated from B. These nodes are disconnected from their original parents and connected to B by vertical expansion.

We present one sharing strategy for set operator queries. Among all sharable nodes, choose the one that will operate with the least amount of data to obtain the final results. For example, when the set operator is UNION, choose the node that contains the most number of tuples. For another example, when the set operator is difference (MINUS) and the sharable nodes will perform further set difference operations to obtain the final results, choose the one that contains the least number of tuples.

1.2 Incremental Evaluation

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

Various stream join operators were proposed. These include stream joins, such as XJoin [117], MJoin [121], and the window join in [55], and stream aggregates, such as Ripple join [64], window aggregates [82], quantile estimates [38], and top-K queries [14]. These operators are not immediately applicable to the ARGUS architecture since it is implemented atop a DBMS. However, when ARGUS migrates to a DSMS, these stream operators are very pertinent, and ARGUS existing incremental evaluation methods should be implemented in the stream operators as well.

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

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

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

The incremental selection and join methods were inspired from the Rete algorithm [50] which stores intermediate results to save repetitive computations in recursive matching of newly produced working elements.

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

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

1.3 Query Optimization

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

Transitivity inference derives implicit highly-selective predicates from existing query predicates to filter out many non-result records in earlier stages and reduce the amount of data to be processed later. The experiments show up to twenty fold improvement for applicable queries; see Section 10.2.2. There are relevant works on inferring hidden predicates [92, 94]. However, they deal with only the simplest case of equijoin predicates without any arithmetic operators. With our canonicalization procedure, ARGUS is able to derive implicit selection predicates from general 2-way join predicates and other selection predicates.

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

The optimizer also decides whether the results of a set of selection predicates will be materialized or not based on selection factors (conditional materialization). This operational choice is implemented based on the observation that when intermediate results are not reduced substantially from the original data, the time saved from the repetitive computations may be offset or exceeded by the materialization overhead (I/O time).

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

1.4 ARGUS Prototype

ARGUS prototype is the integration of the stream techniques we explored. It supports continuous large-scale complex query matching. The shared query plan has the persistent storage and can be incrementally expanded with new queries.

It is built atop a commercial DBMS Oracle. Such choice allows us to focus on the stream-related problems without worrying about implementing the underlying execution engine. Particularly, the goal of the incremental multiple query optimization and other optimization techniques aim at constructing an optimal shared plan, which is largely independent of how the plan is evaluated by the engine. While the engine interface and architecture cast the way how incremental evaluation can be implemented and have significant impact on performance, IMQO and other optimization techniques are complex enough to merit dedicated study. At the beginning of this research, there is no available DSMS engine to work with. Even now there are a few, they do not meet the requirement to serve as an engine base due to the stability issue and the limited support for complex queries.

Using a DBMS also provides us a solid evaluation platform. Particularly, we can evaluate the effects of the developed techniques by comparing them with the naive DBMS solutions. And we can safely evaluate different stream techniques on a mature conventional data query engine.

A more interesting and instant benefit of using DBMS is that ARGUS can be offered as a value-added package to existing database applications where the requirement of stream processing is emerging. We are looking for opportunities to apply the work in national geo-spatial databases, for instance.

While the DBMS is the current choice, our ultimate goal is to make the techniques in practical DSMS systems as well as they also reach a state of maturity. The IMQO framework and the query optimization techniques can be ported to DSMS without modification, but the incremental evaluation methods must be re-implemented based on the engine-specific data structures, such as synopses that store intermediate results.

1.5 An example

In this section, we present two query examples to illustrate the desired sharing and optimization. The queries are formulated on the synthesized FedWire money transfer data (FED), one of the two data sets that we used for system testing and evaluation. FED contains one single data stream. The stream contains money transfer transaction records, one record per transaction. The schema is given in Appendix A.

Consider a query Q_1 on big money transfers for financial fraud detections.

Example 1.1 (Q_1) *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:*

$$\begin{array}{lll}
SELECT & r1.tranid\ r2.tranid\ r3.tranid & \\
FROM & FedWireTrans\ r1, & \\
 & FedWireTrans\ r2, & \\
 & FedWireTrans\ r3 & \\
WHERE & r1.type\_code = 1000 & \textit{—p1} \\
AND & r1.amount > 1000000 & \textit{—p2} \\
AND & r2.type\_code = 1000 & \textit{—p3} \\
AND & r3.type\_code = 1000 & \textit{—p5} \\
AND & r1.rbank\_aba = r2.sbank\_aba & \textit{—p7} \\
AND & r1.benef\_account = r2.orig\_account & \textit{—p8} \\
AND & r2.amount > 0.5 * r1.amount & \textit{—p9} \\
AND & r1.tran\_date <= r2.tran\_date & \textit{—p10} \\
AND & r2.tran\_date <= r1.tran\_date + 20 & \textit{—p11} \\
AND & r2.rbank\_aba = r3.sbank\_aba & \textit{—p12} \\
AND & r2.benef\_account = r3.orig\_account & \textit{—p13} \\
AND & r2.amount = r3.amount & \textit{—p14} \\
AND & r2.tran\_date <= r3.tran\_date & \textit{—p15} \\
AND & r3.tran\_date <= r2.tran\_date + 20; & \textit{—p16}
\end{array}$$

We add two predicates that can be inferred automatically [77] from predicates $p2$, $p9$, and $p14$ by transitivity inference. The inferred predicates are $p4 : r2.amount > 500000$ and $p6 : r3.amount > 500000$. We classify the predicates into PredSets based on the table references:

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

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

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

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

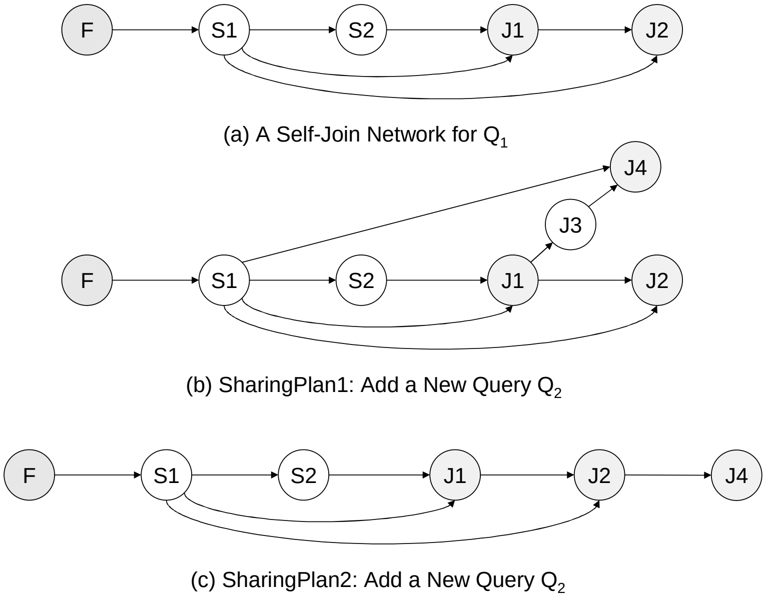
$$\begin{array}{lllll} & r2.rbank\_aba & = & r3.sbank\_aba & AND \\ & r2.benef\_account & = & r3.orig\_account & AND \\ P_5 & r2.amount & = & r3.amount & AND \\ & r2.tran\_date & <= & r3.tran\_date & AND \\ & r3.tran\_date & <= & r2.tran\_date + 20 & \end{array}$$

Assume the selection PredSets (P_1, P_2, P_3) are very selective, thus the optimal evaluation plan will evaluate them first, as shown in Figure 1.1(a). Because PredSets P_2 and P_3 are equivalent, they share the same node $S1$. The results of P_1 are always a subset of that of P_2 or P_3 (or $P_1 \rightarrow P_2$, and $P_1 \rightarrow P_3$), thus P_1 can be evaluated from $S1$ to obtain the results in $S2$ instead of evaluating from the source node $B1$. This improves the performance since much less data needs to be processed to obtain $S2$. 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.

In this example, identifying equivalent PredSets P_1 and P_2 allows them to share the same node $S1$, and identifying the subsumption relation between P_3 and P_1/P_2 allows $S1$ to be used to obtain results of P_3 with reduced computations.

Consider a second query Q_2 which is the same to Q_1 except that 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.

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



(a) A Self-Join Network for Q_1

(b) SharingPlan1: Add a New Query Q_2

(c) SharingPlan2: Add a New Query Q_2

Figure 1.1: Sharing Query Networks. F presents $FedWireTrans$.

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

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 $J3$ as shown in Figure 1.1(b). Then P_7 is evaluated to obtain final results in $J4$. A better sharing choice SharingPlan2 that avoids creating the join node is shown in Figure 1.1(c). 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 to obtain $J4$:

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

From the examples, we see the system needs to do following:

1 Recognize equivalent and subsumed predicates. Particularly, $r2.tran\_date <= r1.tran\_date + 10 \rightarrow r2.tran\_\ date <= r1.tran\_date + 20$. This is achieved by canonicalization, then exact match on the left sides, and comparison on the right side constants.

2 Recognize equivalent and subsumed PredSets. Particularly, $P_2 \equiv P_3$, $P_1 \rightarrow P_2$, $P_1 \rightarrow P_3$, $P_6 \rightarrow P_4$, and $P_7 \rightarrow P_5$. This is achieved by the subsumption identification algorithms.

3 Associate PredSets with plan nodes and the topological connections. This is achieved by searching topological information.

4 Choose the optimal sharing path among multiple ones. In above examples, we want to choose SharingPlan2. This is achieved by applying a sharing strategy, sharing-selection or match-plan.

5 Perform query plan expansion. For example, when we create the new node $J4$, we need to find out the actual PredSet $P_8 = (P_6 \cup P_7) - (P_4 \cup P_5)$. Involving subsumption relations, this becomes more complicated. This is achieved by a set of extended set operations and several updates to the system catalog.

1.6 Thesis Statement and Contributions

The thesis statement is:

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

The thesis contributions are as following.

- **Incremental multiple query optimization:** We design, implement, and evaluate an IMQO framework. It is comprised of the following components.
  - A comprehensive computation indexing scheme to support general plan structures for selection-join-projection queries, aggregate queries, and set operation queries.
  - A set of efficient algorithms to search common computations.
  - Several effective sharing strategies to construct shared query network.
  - A set of tools to incrementally construct the query network and to update the computation index.
- **Incremental evaluation:** We design, implement, and evaluate the incremental evaluation mechanism for selection, join, algebraic aggregates, and set operations.
- **Query optimization:** We explore and evaluate several effective query optimization techniques, including join order optimization, conditional materialization, transitivity inference, and minimum column projection.
- **System implementation:** We build the system atop a DSMS Oracle for direct practical utility for existing database applications where the needs of stream processing become increasingly demanding.
- **Evaluation:** We study and analyze the effectiveness of the implemented techniques. It shows that each technique provides significant performance improvements for general or specific queries, and that the implemented system supports large-scale continuous queries efficiently across different applications.

Chapter 2 discusses the related work. Chapter 3 describes the ARGUS system architecture and the query network structures. Chapter 4 describes the incremental evaluation methods. Chapter 5 describes the query optimization techniques and the optimizer design. Chapter 6 describes computation indexing, searching, and sharing on selection and join queries. Chapter 7 describes computation indexing, searching, and sharing on aggregate and set operation queries. Chapter 8 describes minimum column projection implementation including the column projection indexing, searching, and enrichment. Chapter 9 describes the tools for processing queries and constructing shared query networks. Chapter 10 presents the evaluation results. And Chapter 11 discusses the future work and concludes.

Chapter 2

Related Work

This thesis is related to several database research areas, and bears connections to ideas applied in information retrieval. In this chapter, we describe related work in following areas:

- **Data Stream Management Systems:** The thesis shares the similar goal of providing efficient stream processing functionalities to the recent DSMS prototypes, but is distinguished from them by the extensive IMQO support.

- **Multiple Query Optimization and View-based Query Optimization:** The thesis applies the relational model to index and search common computations. It provides the advantages of fast search, compact storage, and easy update to the indexed computations. The approach is very different from the previous approaches employed in traditional MQO and VQO settings that use linked data structures (query graph and filter tree). The optimal plan search space and heuristics are also substantially different. MQO either applies global search or uses heuristics to construct the plan from scratch, VQO searches the sharable views but not the immediate results, but IMQO search sharable nodes including intermediate results with possible local restructuring.

- **Data Warehousing:** The thesis supports IMQO on multiple aggregate queries. This is related to, but distinguished from data warehousing techniques where multiple aggregate queries are optimized once for all.
- **Incremental Evaluation:** The thesis supports incremental evaluation for stream operators by materializing intermediate results. It is inspired from Rete algorithm used in rule-based production systems. This part is also related to the recent development of stream operators and algorithms, active databases, and materialized view maintenance.
- **Normalization and Indexing in Information Retrieval:** The predicate canonicalization and relational model indexing described in the thesis are inspired from the word normalization and inverted indexing in information retrieval, the common and effective practice to index and search large-scale text collections.
- **Query Optimization:** The thesis implements several query optimization techniques. These are related to traditional query optimization techniques including join-order search and transitivity inference.
- **Other Related Work:** The thesis is also tangentially related to sequence databases, time-series databases, temporal databases, and XML search.

2.1 Data Stream Management Systems (DSMS)

To address the rising challenges of stream processing, several DSMS prototypes have been developed. The well-known ones are Stanford STREAM [89], Berkeley TelegraphCQ [28], and Brandeis/Brown/MIT's Aurora/Borealis [2]. There are also prototypes and systems developed for specific applications. For example, NiagaraCQ [34, 32] and OpenCQ/WebCQ [114] provide Internet content update notification services to large-scale subscribers by matching relevant content changes to a large number of user queries. Gigascope [39] and Tribeca [113] provide platforms to efficiently monitor, query, and analyze network packet streams.

We have witnessed the tremendous scale and complexity of data management from the prosperity of database research and commercial DBMS systems. Such scale and complexity also apply to data stream management. So even the general-purpose DSMS prototypes do not address every prominent component. The one that is often missed or over-simplified is the computation sharing component. This problem becomes more intricate when supporting IMQO, the only practical way to deal with large-scale queries.

Generally, the architectures of all existing DSMS prototypes support sharing among multiple queries. However, due to the complexity of query syntax and semantics, these systems did not fully investigate the approaches to identifying sharable common computations. They either lack or simplify the computation indexing scheme which is vital for IMQO.

The Stanford STREAM [89, 13, 19, 12] is a general-purpose DSMS prototype that employs the traditional modular-processing architecture and constructs tree-shaped query evaluation plans. It supports a declarative stream query language CQL [9] (an extension to SQL), has rich supports for adaptive query processing and load shedding [15], and implements various resource sharing algorithms [8]. However, lacking the computation indexing and searching component, the sharing is limited to the levels of named subqueries (views), and does not yet support sharing across multiple queries.

The Berkeley TelegraphCQ [28, 78, 70] is a general-purpose DSMS prototype that builds around Eddies [10], the integrated adaptive query processer. The query results are obtained by routing tuples to various query operators. TelegraphCQ [84] supports incremental sharing among multiple queries by grouping and indexing individual predicates. [79] shows how sharing can be optimized on adaptive dataflow models, such as on

TelegraphCQ, by avoiding as much repeated work as possible by sharing, yet minimizing waste work caused by sharing. However, the prototype does not address common complex expression identification. And the simplified indexing technique is not suitable for plan-based architecture, such as STREAM, since it does not identify relationship among predicate sets, and does not record topological information of a query network.

The Aurora/Borealis [24, 2, 1] are developed by three universities, Brandeis, Brown and MIT, with rich supports on distributive processing and tolerance of failures. Unlike other DSMSs that use declarative query languages, Aurora/Borealis applies a procedural query language. An Aurora query is a network of connected operator boxes formulated by an administrator, and presents a ready-to-execute plan. 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.

Close to our ARGUS framework, NiagaraCQ [33, 34] supports IMQO. Simple selection predicates are grouped by their expression signatures and evaluated in chains. Equi-join predicates can also be shared. Again it apply only shallow syntactic analysis and limit the matching to the predicate level. This simplified approach also restricts the sharing path selection strategies. NiagaraCQ applies the match-plan strategy. Match-plan matches a plan optimized for the single new query with the existing query network with bottom-up search. This strategy may fail to identify certain sharable computations by fixing the sharing path to the pre-optimized plan. In contrast, ARGUS indexes computation information on predicate set level and topological level, and applies the more flexible sharing strategy, sharing-selection, as well.

Computation sharing is also related to predicate grouping in event-processing model systems such as Gator in Ariel [67, 68] and Trigger Grouping in WebCQ [114]. Both systems implemented the chained sharing on simple selection predicates.

Gigascope [39, 38] is a specialized DSMS for network analysis applications. Nile [65] is a DSMS prototype on distributed sensor network applications. And CAPE [131] is a DSMS prototype concerning dynamic query re-optimization. Tukwila [73] is an adaptive query processing system for Internet applications. IrisNet [54, 42] focuses on multimedia sensor networks. And [7, 45] describe publish/subscribe services. They do not support IMQO.

Alert [105] and Tapestry [115] are two early systems built on DBMS platforms. Alert uses triggers to check query conditions, and uses modified cursors to fetch satisfied tuples. This method is not efficient in handling high data rates and the large number of queries in a stream processing scenario. Similar to ARGUS, Tapestry's incremental evaluation scheme is also wrapped in a stored procedure. However, its incremental evaluation is realized by rewriting the query with sliding window specifications on the append-only relations (streams). This approach becomes inefficient when the append-only table is very large. Particularly, it has to do repetitive computations over large historical data whenever new data is to be matched in joins.

There are also other works on continuous query processing. Query scrambling [118] reschedules the operations of a query during its execution on-the-fly. Rate-based query optimization [122] aims at maximizing the output rate of query plans based on the rate estimates of the streams in the query evaluation tree rather than the sizes of intermediate results. [83] provides a formalism on incremental query evaluation over data changes.

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 [107] and view-based query optimization (VQO). VQO [101, 20, 74] is a query optimization approach that identifies and uses sharable materialized views to speed up the query evaluation. 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 computations, or common subexpressions, are identified by constructing and matching query graphs [48, 27, 31, 129]. 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 [56, 41]. 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.

Foreign-key joins preserve cardinality [56, 129], i.e. joining with a dimension table on the foreign key does not add, duplicate, or drop tuples. Thus a view with foreign-key joins can still be shared by a query even if the query does not have such joins. This cardinality-preserving property is utilized by both [56, 129] to recognize such views. Our prototype currently does not support such findings. However, since foreign-key constraints are orthogonal to the information recorded in the proposed scheme, such feature can be independently implemented and integrated into the prototype without modifying the scheme design.

A sharing strategy specifies the procedure of searching the optimal sharing path in a given search space. In the MQO setting, [107] presents a global search strategy with the A*-search algorithm, and [102] 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.

2.3 Data Warehousing and Aggregates

Efficient aggregation computation has been studied widely for data warehousing and data stream applications. CUBE operator [59] generates aggregates over power sets of aggregating columns (dimensions), and ROLLUP operator generates aggregates over subsets of aggregating columns that are decreasing in the number of dimensions. These operators allow the query optimizer to share the computation from high-dimension aggregate results. GROUPING SETS [4, 69, 100] is a generalization to CUBE and ROLLUP by enumerating the desired subsets of aggregating columns. [35] proposed a hill-climbing approach to search for efficient evaluation plans that share computations among a large number of GROUPING SETS queries. [130] showed how to choose finer-granularity yet-not-requested aggregates that are shared by multiple aggregation queries to improve computation efficiency. [35, 130] also showed that exhaustive search for optimal plans are NP-complete.

Similar to our work, these approaches consider sharing computations among multiple aggregations. Different to ours, these approaches assume all queries are available at the time of optimization and thus perform the optimization as an one-shot operation. Our approach assumes that queries arrive at different times and are added to the existing evaluation plans incrementally in the IMQO framework.

Different from OLAP, data stream aggregates involve constant result updates. Window aggregates, explicitly specifying the stream windows over which the aggregations are performed, however, may cease to receive new tuples as time pass by. Utilizing such property, [82] proposed using Window IDs as an extra aggregating column to avoid buffering tuples. Continuously estimating quantiles and frequent items in distributed streams [38, 85, 14, 91, 86] is another important stream aggregate problem. Since quantiles and frequent items are not algebraic functions, incremental aggregation is not applicable. Their approaches focus on approximate techniques to bound errors and predictive distribution models to minimize communication cost. Orthogonal to above efforts, our work focuses on incremental aggregations for general algebraic functions and support large-scale query systems.

2.4 Incremental Evaluation

The incremental evaluation was inspired from the Rete algorithm which is widely used in rule-based production systems. The Rete algorithm [50] is an efficient method for matching a large collection of patterns to a large collection of objects. By storing partially instantiated (matched) patterns, Rete saves a significant portion of computation that would otherwise have to be re-computed repetitively in recursive matching of the newly produced working elements. Memorizing intermediate results to match new stream tuples is similar to Rete's main idea, and is employed in ARGUS, and other DSMSs.

TREAT [88] and Gator [67, 68] are variants to Rete. TREAT is similar to Rete except that it performs the pattern filtering then match all patterns together. It is similar to perform multi-way joins without materializing any internal join nodes. Gator networks are applied in the Ariel database system to speed up condition testing for multi-table triggers. Gator is a generalization of the Rete and TREAT algorithms. It evaluates selection predicates first, then chooses the join order and selectively chooses which intermediate join results should be materialized. Gator also applies a predicate indexing scheme similar to NiagaraCQ's to evaluate simple selection predicates in chain.

The incremental evaluation is also related to the recent development of stream operators and algorithms, active databases, and materialized view maintenance.

Stream operators are studied in recent literature. XJoin [117], MJoin [121], and the window join in [55] are non-blocking window join operators developed for stream processing. Ripple join [64] is a stream aggregate operator. Window aggregates[82] explore stream/query time constraints to detect the aggregating groups that no longer grow and thus avoid unnecessary tuple buffering. The studies on continuously estimating holistic functions, i.e. quantiles and frequent items, in distributed streams [37, 91] focus on approximate techniques and models to bound errors and to minimize communication cost. These operators are not immediately applicable to the ARGUS architecture since it is implemented atop a DBMS. However, when ARGUS migrates to a DSMS, these stream operators are very pertinent, and ARGUS existing incremental evaluation methods should be implemented in the stream operators as well.

Both active databases and materialized view maintenance concern with the incremental result updates upon data changes. Active database systems [63, 125, 26, 87, 105] allow users to specify, in the form of rules or triggers, actions to be performed upon changes of database states. And materialized view maintenance [20, 74, 60, 97, 51] concerns the techniques of refreshing the views when base relations are changed. While the core algorithms are similar to incremental evaluation methods, the functional mechanism and the purpose are very different. The techniques are not suitable for stream processing since the triggers and the view maintenance updates are hard to share and are not scalable.

2.5 Normalization and Indexing in Information Retrieval

The ideas of predicate canonicalization and computation indexing are inspired from traditional information retrieval techniques including word normalization, dimension reduction, and inverted indexing.

Word normalization converts words into stem forms [126], e.g. *presenting*, *presents*, *presented* are all converted into *present*. Dimension reduction [40] maps various synonyms to the same presentation (a cluster, for example). Canonicalization bears similarity to these techniques. It converts semantically-equivalent yet syntactically-different literal predicates into the same canonical form through legal transformations, such as mathematical transformations and rule-based inference, and allows equivalent and subsumed predicates in different forms to be identified in constant time (when hashing is used).

Building inverted lists to efficiently search large-scale document collections is the fundamental component of a search engine [103]. Beyond term indexing, document structures are also indexed in several research prototypes and commercial search engines (Google). Unlike dealing with the ambiguity of natural languages in information retrieval, the computation information of queries can be exactly described and preserved in a set of indexing tables. The tables should be designed to provide fast searching and easy updating.

2.6 Query Optimization

Traditional query optimization techniques [58], particularly the techniques that search the optimal execution plans [106, 72] based on cost models [95, 71], provide valuable insights to continuous query optimization.

The premises and the assumptions that stream applications run upon vary significantly, which leads to various optimization objectives. For example, TelegraphCQ's Eddy [10] and STREAM's A-Greedy [17] optimize the adaptivity of dynamic query plans to the changing data rates and distributions. Rate-based optimization [11, 122] aims at maximizing output rates. In ARGUS, the optimization goal is minimizing the incremental query evaluation time, which is achieved by minimizing intermediate result sizes by exploiting the very-high-selectivity query property.

There are some relevant works on inferring hidden predicates. [92] discusses inferring hidden join predicates so alternative join orders could be considered in optimization. [94] infers selection predicates as ARGUS does. However, these works deal with only the simplest case of equi-join predicates without any arithmetic operators. ARGUS can infer hidden predicates from general 2-way join predicates with comparison operators and arithmetic operators.

Multiple query optimization (MQO) is systematically presented in [107] as a search problem. [108, 5] showed that MQO with joins is NP-complete. [81] showed that view matching is NP-complete. [102] uses heuristics to reduce search space in MQO. [4, 104] showed that multiple aggregate query optimization is NP-complete. [35] showed that even a simple case, aggregating over single columns, is also NP-complete.

2.7 Other Related Work

There are several database research directions tangentially related to streaming data processing. Sequence databases [110, 109, 111] model the logical ordering of sequence data, and apply the information to the optimization of sequence data queries. Time-series databases [46] are interested in the behaviors of subsequences in a stream of time-ordered data items. Temporal databases [112, 44, 127, 128, 22, 57] stress maintaining temporal versions of databases and their evolution. Index selection [6, 29, 119, 66, 62, 30, 124] targets automatic database tuning. XML search [61, 90, 7, 93, 43] mostly explores variant finite state automata (FSA) to find matches in changing XML files efficiently.

Chapter 3

System Overview

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

As part of the ARGUS project, the ARGUS stream processing system is implemented atop Oracle DBMS for immediate practical utility. The system takes continuous queries specified in SQL, generates shared query evaluation plans, and evaluates plans against stream data. The system is comprised of two components, Query Network Generator (NetGen) and Execution Engine (Engine). The NetGen is the implementation of the IMQO framework and is responsible for generating shared query evaluation plans (query network). The Engine is the Oracle query evaluation engine on which a shared query network is executed as stored procedures.

In this chapter, we first provide an overview of the overall ARGUS project, then we discuss adopting SQL as ARGUS query language, and finally we focus on the ARGUS stream processing architecture.

3.1 Overall ARGUS Project

The purpose of the panorama overview is to illustrate the application context in which the ARGUS stream system is intended to function, and the significant role it plays therein. The ARGUS stream system can also be applied in other important applications, such as business intelligence, scientific exploration, and publish/subscribe services.

The Disruptive Technology Office (DTO) is a funding agency of the Intelligence Community, which has a central mission to help the nation avoid strategic surprise. The Novel Intelligence from Massive Data (NIMD) program is aimed at focusing analytic attention on the most critical information found within massive data - information that indicates the potential for strategic surprise [96].

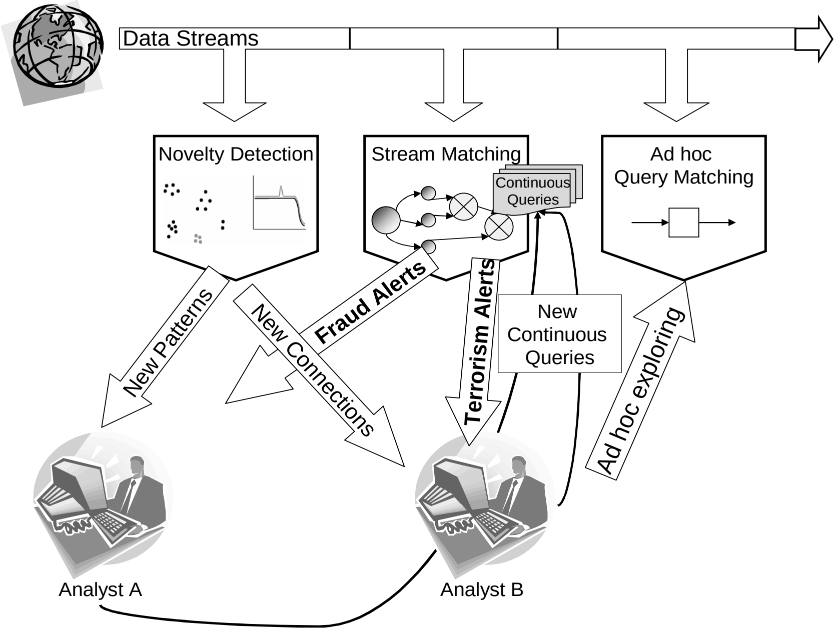


Figure 3.1: ARGUS overall functional architecture. Data streams feed novelty detection, stream matching, and *ad hoc* query matching modules. Novelty detection detects new events and pattern changes, stream matching continuously matches new data with concurrent continuous queries, and *ad hoc* matching efficiently match *ad hoc* queries.

Project ARGUS [23] addresses the problem of obtaining novel intelligence for intelligence analysts from large, constantly incrementing collections of structured data like financial transfers, hospital admission records, or network traffic data. Figure 3.1 shows the overall functional architecture of the ARGUS system. The kinds of analyst task that ARGUS supports include:

- Exploring a large collection of structured data, possibly containing inaccuracies (eg, typos) or being otherwise dirty, using *ad hoc* queries.
- Setting up automated monitoring of streaming data being incrementally added to the collection in order to provide alerts on any of a potentially large set of conditions satisfied by the new data, alone or in combination with data already in the collection.
- Managing the automated analysis of patterns and trends in the data to detect unanticipated, novel events.

The first task is supported by the Dynamix flexible and fast matching software [47] to quickly search through 10^6 to 10^9 records to find the results that match or are close to the *ad hoc* queries. The queries specify complex data selection criteria, and do not have joins and aggregates. The strength of the software is its competitive fast matching capability achieved from the efficient data indexing techniques, and its approximate matching capability achieved from the distance matching techniques.

The third task is supported by Carnegie Mellon innovative cluster analysis techniques for novelty detection [53]. Data points are clustered based on relevant features, then new data points are added to the existing clusters or form new clusters as they arrive. A cluster is characterized by the cluster distance function, a function of the distance of the data points to the cluster centroid. By tracking changes in the radial cluster density function over time, the system can find new clusters and detect changes in both the shape and density of clusters corresponding to unanticipated, novel events.

The second task is supported by the ARGUS stream processing system, the thesis work. Complex continuous queries are formulated by analysts or by the automatic novelty detection tools, and evolve over time. Continuous stream monitoring allows analysts to quickly respond to emergency situations and discover unusual developments of special events. The success lies in fast concurrent continuous query matching over frequently-incrementing data streams, and is achieved by the techniques that will be described in this thesis. In the remaining of the thesis, ARGUS refers to the ARGUS stream processing system (not to tasks 1 and 3), unless otherwise clarified.

3.2 ARGUS Query Language

ARGUS query language is SQL. With a little concept enrichment, SQL is capable of expressing continuous queries. In a DBMS, a relation is a set of data of the same schema. A query (a set of relation operators) operates on a set of relations, and the result of the query is a new relation. In a DSMS, the more important data entities are streams. A stream is a data pipeline of the same schema which keeps on streaming new data tuples. A query that operates on streams (or a combination of streams and relations) apparently should produce results continuously. Thus the query is called *continuous query*, and its result is a new stream. By embracing the concept that continuous queries operate on streams and produce new streams, SQL is capable of expressing semantics of continuous queries.

Due to the long-lived nature on unbounded streams, continuous queries usually concern with the results falling into certain time windows. SQL can express the semantics of time windows. For example, when a query joins two tuples that fall in a time window of fixed-length, the time window can be expressed as a predicate, $0 \leq t2.timestamp - t1.timestamp \leq W$. For another example, the sliding window with regard to the current time can be expressed as a predicate involving the current time function, $sysdate - t1.timestamp \leq W$. The current implementation does not support sliding windows with regard to the current time. This is because the DBMS applies bag semantics to tuples, and thus it is hard to implementing the mechanism of tuple expiration. One roundabout solution is to project the timestamp column all the way along the query network evaluation paths. Then at each time tick, expired tuples and intermediate results can be dropped from the query network.

SQL as the continuous query language provides the compatibility to traditional DBMS interface. Such compatibility is desirable in the applications where both stream processing and traditional data management are important. For example, an intelligence analyst who studies a complicated phenomenon needs to explore a large historical data archive to formulate, refine, and validate his hypothesis with *ad hoc* queries which can be optimally evaluated with an OLAP DBMS or the Dynamix matcher. When an *ad hoc* query is validated as the one to describe an interesting pattern worth tracking, the analyst registers it with the DSMS as a continuous query to monitor new data.

SQL as the continuous query language also provides a way to compare the stream techniques and the DBMS, where streams are treated as relations.

STREAM [9] and TelegraphCQ [28] propose new extensions to SQL to form the continuous query languages. These extensions are mainly adding explicit syntax structures for time window specifications and adding stream-relation conversion operators. As we see from above, SQL can express them with range predicates and the stream-producing assumption. ARGUS can be extended to support new syntax structures in future.

ARGUS assumes that query conditions are in conjunctive normal forms (CNF). Particularly, predicates in where-clause and having-clause are in CNF. This is also a common practice in query optimization and in other stream systems (TelegraphCQ, NiagaraCQ, STREAM). A query condition is a conjunction (connected by AND) of one or more conjuncts, each of which is a disjunction (connected by OR) of one or more literal predicates. We also call the disjunctions *OR predicates (ORPreds)*, and the conjunctions *predicate sets*

(PredSets).

The current implementation supports incremental evaluation and IMQO for queries with selections, joins (including self-join), projections, aggregates, set operators, and view definitions and references.

3.3 ARGUS Stream Processing System

ARGUS contains two components, Query Network Generator (NetGen) and Execution Engine (Engine), shown in Figure 3.2. The system works as following. An analyst sends a request to ARGUS to register a new query Q; NetGen analyzes the query, constructs a new shared optimal query evaluation plan (also called query network) from the existing one, instantiates the plan and generates the updated initialization and execution code, records the plan information in the system catalog, and outputs the updated code; Engine runs the query network to match with data streams and returns the results to the analyst. Section 3.4 overviews the shared plan structures and describes how query networks are evaluated by the execution engine. Section 3.5 overviews the NetGen architecture and how a new query Q in SQL is translated into a part of the shared query network.

3.4 Execution Engine

The engine is the underlying DBMS query execution engine. We use its primitive relation-operator support, but not its complex query optimization functionality, to evaluate the query network generated by ARGUS to produce stream results. As we know, to run a query in SQL, a DBMS generates an optimal logical evaluation plan, then instantiates it to a physical plan, and executes the physical plan to produce the query results. The logical plan can be viewed as a formula comprised of relation operators on the querying relations. And the physical plan specifies the actual methods and the procedure to access

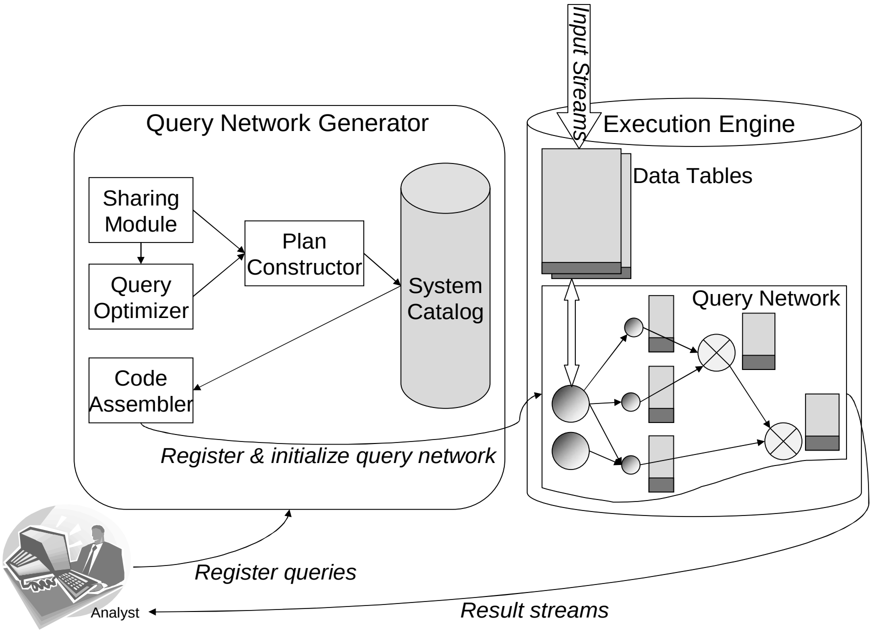


Figure 3.2: Using ARGUS. An analyst registers a query with ARGUS. ARGUS Query Network Generator processes and records the query in the System Catalog, and generates the initialization and execution code. ARGUS Execution Engine executes the query network to monitor the input streams, and returns matched results. The analyst may register more queries.

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, and discuss the design decision on using the DBMS Oracle. We will not discuss how the DBMS generates and executes the physical plans.

3.4.1 Query Network Structure

A query network is a directed acyclic graph (DAG). Figure 3.3 shows an example, which evaluates four queries. The upper part evaluates queries Q_1 and Q_2 described in Section 1.5, 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.

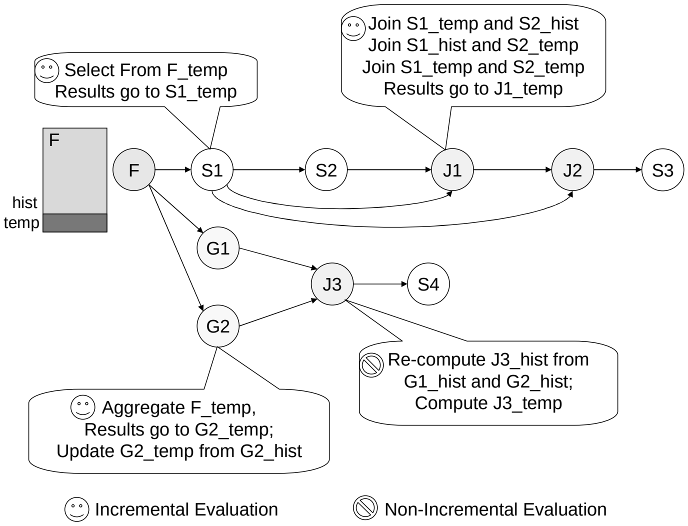


Figure 3.3: Execution of a shared query network. Each node has a historical (hist) table and a temporary (temp) table, here only those of node F are shown. The callouts show the computations performed to obtain the new results that will be stored in the nodes' temporary tables. S nodes are selection nodes, J nodes are join nodes, and G nodes are aggregate nodes. The network contains two 3-way self-join queries and two aggregate-then-join queries, and nodes J2, S3, J3, and S4 present their results respectively.

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

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

An arrow between nodes presents the evaluation of a set of operators on the parent node(s) to obtain the results stored in the child node. The operator 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 3.1 shows the types of operator sets and their result nodes.[1]

Operator Set	# of parents	Result Node Type
Selection PredSet	1	Selection node
Join PredSet	2	Join node
GROUPBY Expressions	1	Aggregate node
Set Operator	≥ 2	Set operator node

Table 3.1: 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 produced solely from the parent's temporary table, e.g. in Figure 3.3, $S1\_temp$ 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 3.3, $J1\_temp$ can be obtained by three joins, $S1\_temp \bowtie S2\_hist$, $S1\_hist \bowtie S2\_temp$, and $S1\_temp \bowtie S2\_temp$. Performing the 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

[1] The system only supports 2-way joins now. The major challenge on supporting multi-way join lies in the NP-hardness of join sharing selection, not in the indexing and searching. We plan to extend to support multi-way joins with local sharing strategies.

much smaller than the historical tables, $|S1\_temp| \ll |S1\_hist|$, and $|S2\_temp| \ll |S2\_hist|$ [77].

The incremental evaluation on selections and joins is inspired from Rete, a fast pattern matching algorithm widely used in production systems. It avoids repetitive computations in the recursive matching process by storing intermediate results.

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 3.3, $G2\_temp$ can be obtained by aggregating F_temp tuples and then adding the old aggregate values from $G2\_hist$ to the aggregate values [76].

Some operator sets can not be incrementally evaluated. Examples include holistic aggregates, such as quantiles [59], 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 3.3. 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.

Regardless a node can be incrementally evaluated or not, the way to populate its temporary table can be expressed by a set of simple SQL queries operating on its parent nodes and/or its own historical table. In another word, the incremental or non-incremental evaluation methods to populate a node's temporary table can be instantiated by simple SQL queries.

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

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

3.4.2 Atop DBMS Oracle

As we see from above, built atop Oracle, the ARGUS engine is a wrap-up of the Oracle engine. It drives the Oracle engine to produce stream evaluation results. The interface between the Oracle engine and the ARGUS engine is the PL/SQL language, a procedural language supported by Oracle to manage the database. The shared incrementally-evaluated query network is realized by stored procedures written in PL/SQL.

Although ARGUS is built atop Oracle, it is not intrinsically clung to Oracle or DBMSs. The query network has its internal presentations and persistent storage in the system catalog. Only the instantiation of the network evaluation is platform-specific. Particularly, the code blocks are instantiated from several platform-specific operator code templates that implements the incremental or non-incremental evaluation algorithms, and the wrap-up stored procedures are generated based on a platform-specific procedure construction template. Thus by just mapping the slight syntax differences between PL/SQL and the procedural language supported by another DBMS, e.g. TSQL by Microsoft SQL Sever, in the code templates and the procedure template, ARGUS can support stream monitoring on that DBMS as well.

Further, ARGUS can also be expanded to work with DSMSs. This will be a major direction of the future work. There are two approaches of coupling ARGUS with a DSMS. The first is the **loose coupling**, where ARGUS sends a DSMS-compliant evaluation plan of the query network or plan updates to DSMS, and DSMS continuously runs the network to produce new results. This is simple and presents an one-way communication from ARGUS to the DSMS. The second approach is the **tight coupling**, where ARGUS is also responsible for adaptive re-optimization of the query network when sub-optimal behavior is detected upon new data trends. This will be a major direction of the future work.

3.5 Query Network Generator

ARGUS Query Network Generator (NetGen) is responsible for generating and updating the shared query evaluation plan (query network). It takes a new query Q, constructs an new optimal shared evaluation plan, expands the existing query network with the plan (plan instantiation), and outputs the updated query network initialization and execution code. Figure 3.4 shows the architecture of NetGen, and illustrates the query processing procedure and information flow between modules.

ARGUS Manager in NetGen is a master program that takes the query in SQL, and invokes different modules to generate the shared plan, update the system catalog, and generate the updated code.

The parser parses the SQL query into a parse tree based on a publicly available SQL grammar.

The canonicalizer is a set of preprocessing programs to convert the parse tree to the standard logical parse tree that is assumed by other processing modules. It performs the transitivity inference (see Section 5.1), classifies the where-clause predicates into PredSets based on the tables they reference, canonicalizes expressions and predicates (see Section 6.2.1), and converts the parse tree into the standard parse tree, such as presenting a conjunct as a strict OR predicate tree even if the conjunct does not contains OR. See Section 6.2 for details.

The incremental multi-query optimizer (IMQO) is comprised of three submodules to process selection-join predicates in where-clause, aggregates in groupby-clause, and set operators (UNION, MINUS, etc.) that connect multiple terms, respectively. The submodules work in a similar approach. A submodule extracts sub-structures from the logical parse tree, uses the index & search interface to search for the matches of the existing query network from system catalog, selects the optimal ones to construct the local optimal sharing plan, and calls the plan instantiator to instantiate the sharing plan and rewrite the logical parse tree to reference the shared nodes. Recursively, IMQO works on the rewritten logical parse tree until no more sharing is possible. Chapters 6, 7, and 9 describe the related details.

The query optimizer generates an optimal evaluation plan for the remaining unsharable logical parse tree. The plan is also called **construction plan** since it expands the existing query network. In the current implementation, the optimizer mainly concerns with join orders and the choice of materializing selection PredSets. Optimizations on aggregates and set operators are not considered. Chapter 5 describes the details.

Correspondingly, the plan instantiator, index & search interface, and query rewriter, all have separate submodules to process where-clause, groupby-clause, and set operators.

The plan instantiator traverses and instantiates a plan (a local sharing plan or an construction plan). It breaks the plan into sub-plans, creates the nodes and generates the code blocks, and calls the index & search module to update the system catalog to reflect such changes. See Chapter 9 for details.

The index & search interface is a set of programs that creates, updates, and searches the index of the query network. The index is stored in the system catalog as a set of relational tables. The common computations are matched and formulated conceptually between sub-structures of the logical parse tree and the index through the interface. Section 3.6 describes the system catalog, and Chapters 6, 7, 8 and 9 describe the indexing and searching various types of query network information.

The query rewriter rewrites the logical parse tree to reference the shared or newly created nodes. The rewritten logical parse tree becomes simpler and simpler as computations done by the newly referenced nodes are dropped. Beyond the submodules to support where-clause, groupby-clause, and set operators, the query rewriter has one more submodule which deals with view definitions and references. See Section 9.2.3 for details.

The code assembler retrieves the initialization code and execution code blocks of all query network nodes, sorts them in the order that complies to the evaluation precedence, then concatenates the initialization codes, and wraps-up the execution code blocks in a set of stored procedures. The evaluation precedence states that an ancestor node must be evaluated before its descendants. See Section 9.3 for details.

Beyond join ordering and conditional materialization, there are two additional optimization techniques applied in NetGen, namely, transitivity inference and minimum column projection.

Transitivity inference is a part of the canonicalizer and derives implicit highly-selective predicates from existing query predicates to filter out many non-result records in earlier stages and reduce the amount of data to be processed later. See Section 5.1 for details.

Minimum column projection is a part of the plan instantiator and projects only necessary columns in the materialized table pairs when a new node is created, to save materialization space and execution time. These columns include those in the final results and those needed for further evaluation. When a node is shared, it may not contain all the columns needed for the new query. Then extra columns will be added to the node and possibly to its ancestors. This process is called *projection enrichment*. See Sections 8.2 and 8.3 for details.

3.6 System Catalog

The system catalog is a set of relations that records the query network information. The information serves two purposes. First, it is used to instantiate the query network execution plan, or say construct the executable code of the query network. Second, it is used to search the common computations between the query network and the new query Q. The system catalog information is updated when the query network is changed, which we call **system catalog synchronization**.

The system catalog relations are categorized into five groups, query network storage, coding storage, query storage, meta storage, and admin storage, shown in Figure 3.5.

Query network storage relations are categorized into three sub-groups in two ways. First by information types, they are categorized into predicate/PredSet/expression indexing information tables, projection information tables, and topology information tables. Second, by node types, they are categorized into selection/join node tables, aggregate node tables, and set operation node tables. These relations record all the query network information that is used for constructing the executable code and searching for common computations. Chapter 6 and 7 describe the details.

Coding storage contains one relation called LinearNodeTable. It records the initialization code and execution code blocks of each node. It also contains the sequencing information of each node, which is used to sort the code blocks when wrapping up them into stored procedures. See Section 9.3.

Query storage contains two relations, QueryTable, and QuerySemanticTable. QueryTable records the registered query information, including query identifiers, the result table of the query, and the registering time, etc. QuerySemanticTable records the original SQL query texts of registered queries.

Meta storage contains four relations. It records the registered streams, user-defined aggregation information and rules (Section 4.2), and the counters that are used for constructing the query network entity names.

Admin storage contains one relation, CheckPointTable. It records the query network checkpoint information. A checkpoint is a dump of query network storage, coding storage, and query storage, and can be used for reconstructing the query network and the system catalog. The CheckPointTable and the related checkpoint utilities are implemented to facilitate the system testing and evaluation, and will also be useful for recovery in real use.

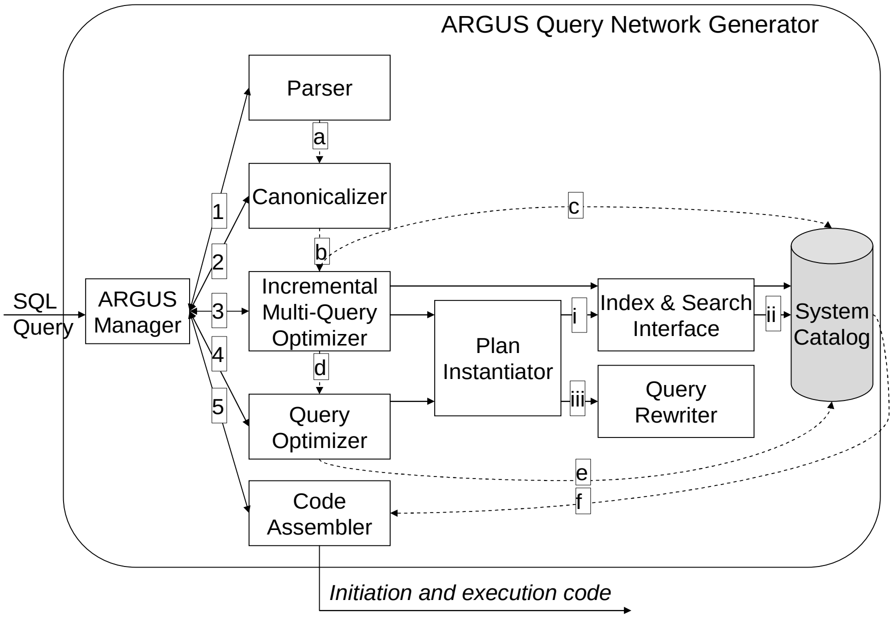


Figure 3.4: Architecture of ARGUS Query Network Generator.

ARGUS Manager receives a query Q in SQL, and invokes sub-components to register the query:
1. Parser parses the query to a parse tree.
2. Canonicalizer converts the parse tree to logical parse tree, where predicates are canonicalized and grouped into PredSets.
3. Sharing Module searches the common computation between the logical parse tree and the query network R, constructs a sharing plan (a selected local optimal sharing path), and calls the Plan Instantiator.
4. Query Optimizer generates an optimized plan for the remaining logical parse tree, and calls the Plan Instantiator.
5. Code Assembler reads nodes' registration information and assembles the code blocks into executables (initialization code and execution code).

Plan Instantiation:
i) Plan Instantiator traverses the plan, and sends individual node create/update instructions to Index & Search Interface.
ii) Index & Search Interface constructs and executes a set of commands to create/update the query node registration in the System Catalog.
iii) Query Rewriter rewrites the logical parse tree.

Information Flow:
a. Parse tree
b. Logical parse tree
c. Sharing plan
d. Remaining logical parse tree
e. Optimized plan for the remaining logical parse tree
f. Node Registration information

Query Network Storage

Indexing Tables:

PredIndex*
PSetIndex*
GroupExprIndex**
GroupExprSet**

Projection Tables:

JoinSimpleColumnNameMap*
JoinExprColumnNameMap*
SelSimpleColumnNameMap*
SelExprColumnNameMap*
GroupColumnNameMap**
UnionColumnNameMap***

Topology Tables:

SelectionTopolgy*
JoinTopology*
GroupTopology**
UnionNode***
UnionTopology***

Coding Storage

LinearNodeTable

Query Storage

QueryTable
QuerySemanticTable

Meta Storage

BaseTableTable
AggreBasicTable
AggreRuleTable
ValueTable

Admin Storage

CheckPointTable

Figure 3.5: System Catalog. * selection/join node, ** aggregate node, *** set operator node.

Chapter 4

Incremental Evaluation

This chapter describes the incremental evaluation algorithms for selections, joins, aggregates, and set operators.

As discussed in Section 1.2, incremental evaluation refers to efficient continuous query evaluation methods over stream data to produce new results. The main idea is minimizing the access to the relatively large-volume historical data. This usually requires materializing intermediate results on historical data.

Different query operators, i.e. selections, joins, aggregates, and set operators, entail different intermediate results and different rules to compute new results. Incremental selection does not require any historical data and computes new results immediately from the new stream data. Incremental join needs to join historical data with new stream data to produce new results. Incremental aggregation needs to store basic aggregate function values on historical data and compute new aggregate values from the basic values and new data. And some set operators need to access historical data to compute new results while others do not.

This chapter describes the incremental evaluation algorithms for selections, joins, aggregates, and set operators.

4.1 Selection and Join: Rete Algorithm

Consider selections and joins on tables N and M. Let n and m denote the historical data, and Δn and Δm the new much smaller incremental data, respectively. By Relational Algebra, a selection operation σ on data $n+\Delta n$ is equivalent to $\sigma(n+\Delta n) = \sigma(n)+\sigma(\Delta n)$. $\sigma(n)$ is the set of the historical results that is materialized. To evaluate incrementally, only the computation on Δn is needed ($\sigma(\Delta n)$).

Similarly, for a join operation $\bowtie$ on $(n + \Delta n)$ and $(m + \Delta m)$, we have $(n + \Delta n) \bowtie (m + \Delta m) = n \bowtie m + \Delta n \bowtie m + n \bowtie \Delta m + \Delta n \bowtie \Delta m$. $n \bowtie m$ is the set of the historical results that is materialized and by far the largest part of the computation since $m \gg \Delta m$ and $n \gg \Delta n$, but needs not to be recomputed. Only the computations on $\Delta n \bowtie m + n \bowtie \Delta m + \Delta n \bowtie \Delta m$, three small joins, are needed. Since Δn and Δm are much smaller than n and m, the time complexity of the incremental join is linear to $O(n+m)$. In the implementation, the incremental selection or join is executed only when the corresponding new data part contains new data (not empty) based on the Boolean flag value of the corresponding node's parent. For example, $\Delta n \bowtie m$ is only executed when Δn is not empty.

The incremental evaluation algorithms on selections and joins are inspired from Rete, a fast pattern matching algorithm widely used in production systems. It avoids repetitive computations in the recursive matching process by storing intermediate results.

Figure 4.1 shows the optimal query network for Example 1.1. And Figure 4.2 shows how the PredSets are incrementally evaluated to obtain the new results for Example 1.1 through the Rete algorithm.

Using the Rete algorithm on selection and join queries leads to up to 10-fold performance improvement comparing to using the original SQL queries on the DBMS directly; see Section 10.2 for evaluation details.

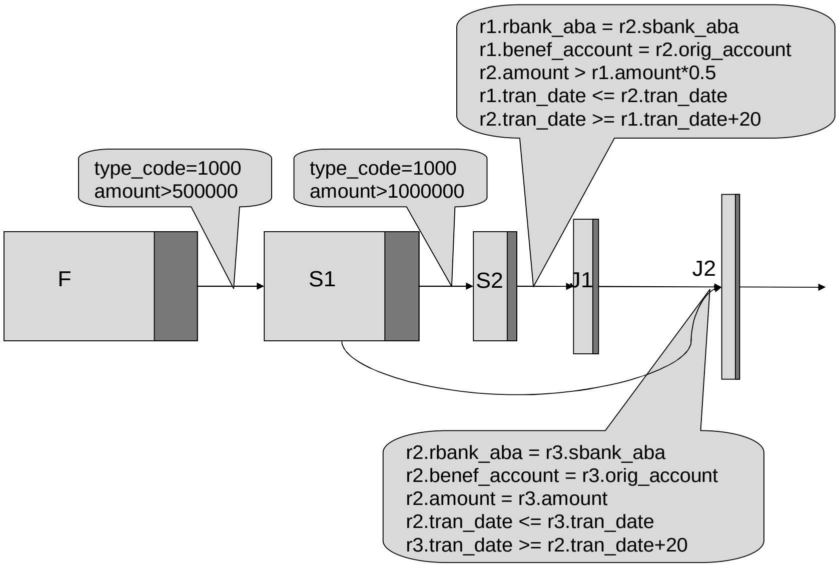

r1.rbank_aba = r2.sbank_aba
r1.benef_account = r2.orig_account
r2.amount > r1.amount*0.5
r1.tran_date <= r2.tran_date
r2.tran_date >= r1.tran_date+20
type_code=1000
amount>500000
type_code=1000
amount>1000000
F
S1
S2
J1
J2
r2.rbank_aba = r3.sbank_aba
r2.benef_account = r3.orig_account
r2.amount = r3.amount
r2.tran_date <= r3.tran_date
r3.tran_date >= r2.tran_date+20

Figure 4.1: Optimal query network for Example 1.1.

4.2 Incremental Aggregation

Aggregate functions can be classified into three categories based on whether and how the aggregate value on the whole data set S can be computed from its partition $\{S_j|j = 1, ..., K\}$ [59]; see below. ARGUS handles all three types of aggregates, and performs incremental aggregation on both distributive and algebraic aggregates even if they are user-defined aggregates.

Distributive: Aggregate function F is distributive if there is a function G such that $F(S) = G(F(S_j)|j = 1, ..., K)$. COUNT, MIN, MAX, SUM are distributive.

Algebraic: Aggregate function F is algebraic if there is a function G and a multiple-valued function F' such that $F(S) = G(\{F'(S_j)\}|j = 1, ..., K)$. AVERAGE is algebraic with $F'(S_j) = (SUM(S_j), COUNT(S_j))$ and $G(F'(S_j)|j = 1, ..., K) = \frac{\sum F'_1(S_j)}{\sum F'_2(S_j)}$ where F'_i is F''s ith value.

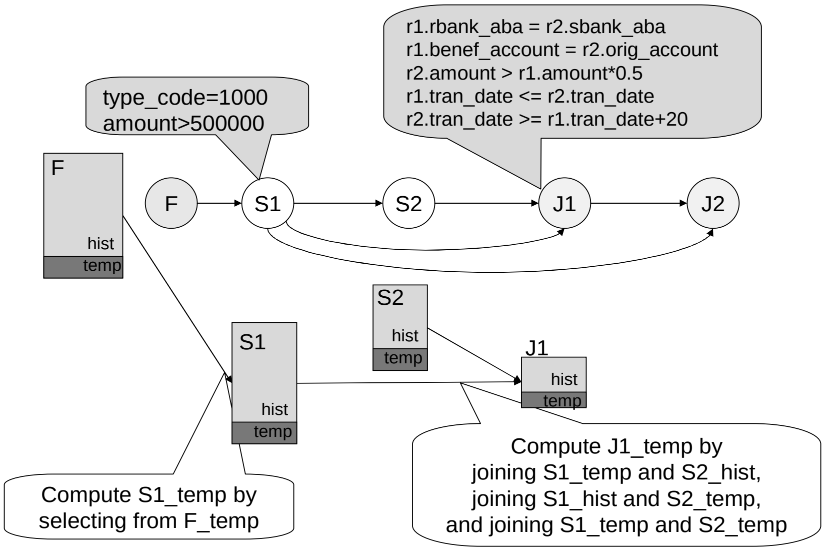

r1.rbank_aba = r2.sbank_aba
r1.benef_account = r2.orig_account
r2.amount > r1.amount*0.5
r1.tran_date <= r2.tran_date
r2.tran_date >= r1.tran_date+20
type_code=1000
amount>500000
F
hist
temp
F
S1
S2
J1
J2
S2
hist
temp
S1
hist
temp
J1
hist
temp
Compute J1_temp by
joining S1_temp and S2_hist,
joining S1_hist and S2_temp,
and joining S1_temp and S2_temp
Compute S1_temp by
selecting from F_temp

Figure 4.2: Incremental evaluation of the optimal query network for Example 1.1. It illustrates the evaluation of node $S1$ and $J1$. $S1$ is the results of a selection PredSet $\{type\_code = 1000\ AND\ amount > 500000\}$, and is incrementally evaluated by performing the selection from F_temp to obtain S1_temp. $J1$ is the results of a join PredSet $\{r1.rbank\_aba = r2.sbank\_aba,\ \ AND\ ...\}$, and is incrementally evaluated by performing the three small joins from $S1$ and $S2$ to obtain J1_temp.

Holistic: Aggregate function F is holistic if there is no constant bound on storage for describing F'. Quantiles are holistic.

Distributive and algebraic functions can be incrementally updated with finite data statistics while holistic functions can not. To perform incremental aggregation for arbitrary distributive/algebraic functions including user-defined ones, we use two system catalog tables to record the types of the necessary statistics and the updating rules, shown in Tables 4.1 and 4.2. Table *AggreBasics* records the necessary bookkeeping statistics for each distributive/algebraic function. Argument X in *BasicStatistics* indicates the exact match when binding with the actual value while W indicates the wild-card match. Table *AggreRules* records incremental aggregation rules. The rule of a distributive function specifies how the new aggregate is computed from the historical data (S_H) and the temporary data (S_N); and the rule of an algebraic function specifies how the new aggregate is computed from the basic statistics.

AGGREGATE FUNCTION	AGGREGATE CATEGORY	INCREMENTAL AGGREGATION RULE	VERTICAL EXPANSION RULE
AVERAGE	A	$SUMX/COUNTW$	$SUMX/COUNTW$
SUM	D	$SUMX(H) + SUMX(N)$	$SUM(SUMX)$
MEDIAN	H	NULL	NULL
COUNT	D	$COUNTW(H) + COUNTW(N)$	$SUM(COUNTW)$

Table 4.1: AggreRules. Aggregate Category: A Algebraic; D Distributive; H Holistic.

AGGREGATE FUNCTION	BASIC STATISTICS	BASIC STATID
AVERAGE	$COUNT(W)$	$COUNTW$
AVERAGE	$SUM(X)$	$SUMX$
SUM	$SUM(X)$	$SUMX$
COUNT	$COUNT(W)$	$COUNTW$

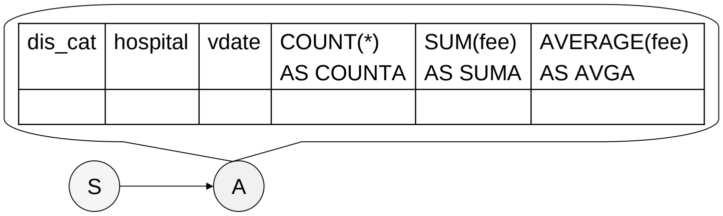
Table 4.2: AggreBasics

Consider query A that monitors the number of visits and the average charging fees on each disease category in a hospital everyday, shown in Example 4.1. When new tuples from the stream Med arrive, the aggregates *COUNT(*)* and *AVERAGE(fee)* can be incrementally updated if *COUNT(*)* and *SUM(fee)* are stored, shown in Figure 4.3. Node S presents the data stream Med, and node A presents query A's results.

Example 4.1 (A) *Monitoring the number of visits and the average charging fees on each disease category in a hospital everyday.*

$$\begin{array}{ll}
SELECT & dis\_cat, hospital, vdate, \\
 & COUNT(*), AVERAGE(fee) \\
FROM & Med \\
GROUP\ BY & CAT(disease)\ AS\ dis\_cat \\
 & hospital, \\
 & DAY(visit\_date)\ AS\ vdate
\end{array}$$

Incremental aggregation is shown in Algorithm 4.1 and is illustrated by Figure 4.4 on $AVERAGE(fee)$ for query A.



dis_cat	hospital	vdate	COUNT(*) AS COUNTA	SUM(fee) AS SUMA	AVERAGE(fee) AS AVGA

(a) Evaluate query A

Figure 4.3: Evaluating Query A.

Algorithm 4.1 also shows the time complexity T_i of each step. The merge step (Step 2) is realized with a hash join on A_H and A_N with $T_{hash2} = O(|A_H| + |A_N|)$. If the A_H hash is precomputed and maintained in RAM or on disk with perfect prefetching, the time complexity is $T_{prefetch2} = O(|A_N|)$. The duplicate-drop step (Step 4) is realized by a set difference $A_H - A_N$, which can be achieved by hashing with the time complexity of $T_{hash4} = O(|A_H| + |A_N|)$ or by prefetched hashing with the time complexity of $T_{prefetch4} = O(|A_N|)$. However, we observed that it took $T_{curr4} = O(|A_N| * (|A_N^H|)) = O(|A_N|^2)$ on the DBMS, where $|A_N^H|$ is the number of the groups in S_H to be dropped. This means that the DBMS applies the nested-loop algorithm to compute the set difference.

If the incremental aggregation is implemented in a DBMS or a DSMS as a built-in operator, both merge and duplicate-drop steps can be achieved with hashing. With the prefetching, the complexity will be linear to $|A_N|$. Therefore, the time complexities on the current implementation, build-in operator with hashing, and with prefetch are following: $T_{curr} = O(|S_N^2| + |A_H|)$, $T_{built-in} = O(|S_N| + |A_H|)$, and $T_{prefetch} = O(|S_N|)$.

Algorithm 4.1 ***Incremental Aggregation***

0. PredUpdate State. *A_H contains update-to-date aggregates on S_H.*

1. ***Aggregate*** *S_N, and put results into A_N. $T_1 = O(|S_N|)$*

2. ***Merge groups*** *in A_H to A_N. $T_{hash2} = O(|A_H| + |A_N|)$, $T_{prefetch2} = O(|A_N|)$*

3. ***Compute algebraic aggregates*** *in A_N from basic statistics (omitted for distributive functions). $T_3 = O(|A_N|)$*

4. ***Drop duplicates*** *in A_H that have been merged into A_N.*

$T_{curr4} = O(|A_N| * |A_N^H|) = O(|A_N|^2)$,

$T_{hash4} = O(|A_H| + |A_N|)$, $T_{prefetch4} = O(|A_N|)$

5. ***Insert new results*** *from A_N to A_H, preferably after A_N has been sent to the users. $T_5 = O(|A_N|)$*

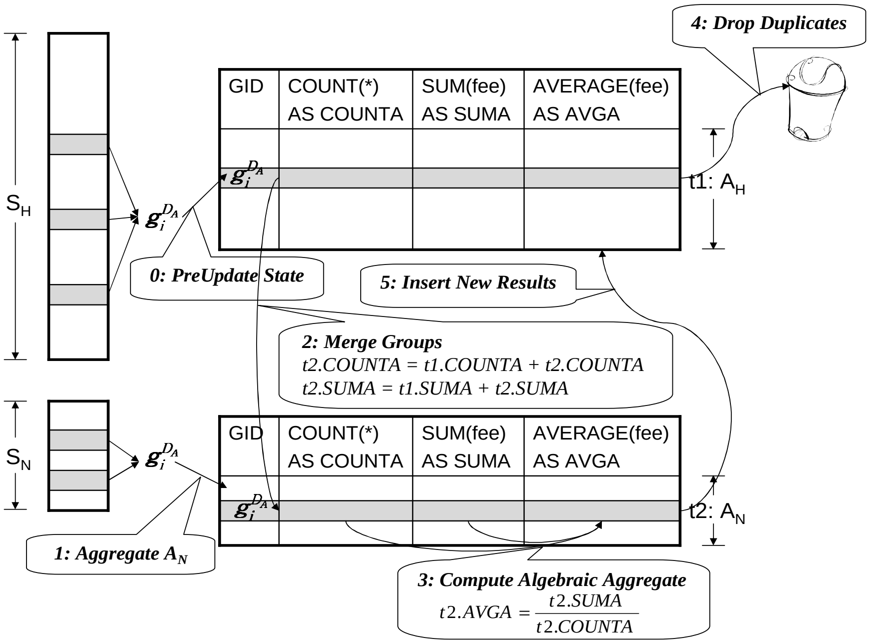


Figure 4.4: 5-step Incremental Aggregation.

Figure 4.5 shows the procedure to instantiate the incremental aggregation code for function $AVERAGE(fee)$ in query A.

1. The function is parsed to obtain the function name and the list of the actual argu-

ments.

2. The basic statistics and updating rules are retrieved.
3. The statistics are parsed and their formal arguments are substituted by the actual arguments.
4. The statistics are renamed and stored in *GroupColumns*.
5. The name mapping is constructed based on above information.
6. The updating rules are instantiated by substituting formal arguments with the renamed columns.

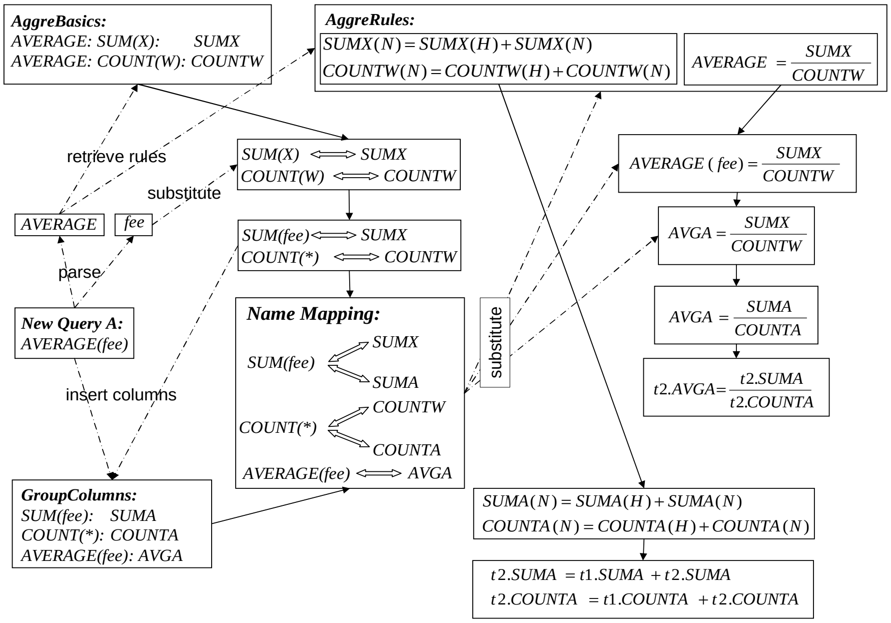


Figure 4.5: Incremental aggregation instantiation. Generate the incremental aggregation code by rewriting incremental aggregation rules with actual arguments.

Using the incremental aggregation on algebraic aggregate queries shows up to 100-fold performance improvement comparing to using the regrouping approach; see Section 10.3 for evaluation details.

4.3 Set Operators

This section describes the incremental evaluation on set operators. We consider three set operators, UNION ALL, UNION, and MINUS.

Set operator, UNION ALL ($+$), takes data elements as a bag and outputs all data elements from the operand relations/streams, as shown in Figure 4.6. Thus, we have $(m + \Delta m) + (n + \Delta n) = (m + n) + (\Delta m + \Delta n)$. The incremental evaluation is easy by simply unioning all new data parts.

Set operator, UNION ($\cup$), takes data elements as a set (identical data elements present just once) and outputs all distinct data elements from the operand relations/streams, as shown in Figure 4.7. Thus, we have $(m + \Delta m) \cup (n + \Delta n) = (m \cup n) \cup (\Delta m \cup \Delta n)$. Therefore, the incremental evaluation is easy by unioning new data parts. However, the results may contain duplicates that are already in $m \cup n$, which are not needed by the user. Duplicates can be dropped by differentiating the results from $m \cup n$. We have $(m + \Delta m) \cup (n + \Delta n) = (m \cup n) \cup (\Delta m \cup \Delta n) = (m \cup n) + ((\Delta m \cup \Delta n) - (m \cup n))$.

Set operator Difference (-) is similar to a join, as shown in Figure 4.8. We have $(m + \Delta m) - (n + \Delta n) = (m - n) + (m - \Delta n) + (\Delta m - n) + (\Delta m - \Delta n)$. However, continuous queries usually have a time-segregate property. The time-segregate property holds on a set-operation query if and only if for any two non-overlapped time period τ_1 and τ_2, the corresponding data parts of the query do not overlap, that is $\Delta m_{\tau_1} - \Delta n_{\tau_2} = \phi$ and $\Delta m_{\tau_2} - \Delta n_{\tau_1} = \phi$. This property clearly holds when streams keep on streaming in unique new data tuples (may be unique just on timestamps). All the queries that involve the difference operator in our query repository (simulation of intelligence analysis queries) have the property. So we always have $(m - \Delta n) = \phi$ and $(\Delta m - n) = \phi$. Therefore, the incremental evaluation can be simplified as $(m + \Delta m) - (n + \Delta n) = (m - n) + (\Delta m - \Delta n)$.

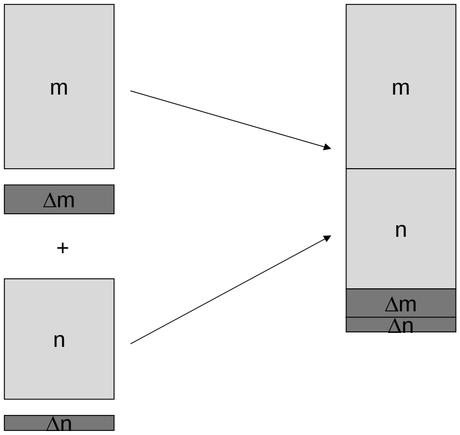


Figure 4.6: Incremental evaluation for UNION ALL

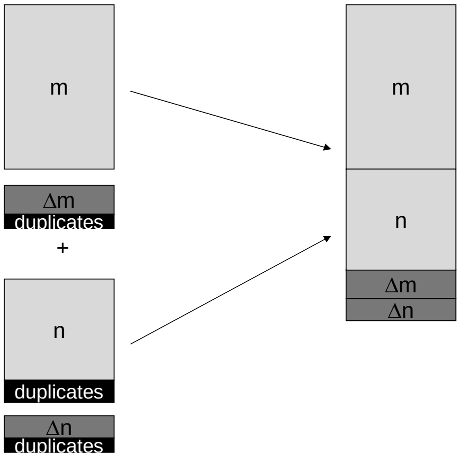


Figure 4.7: Incremental evaluation for UNION. Duplicates are dropped.

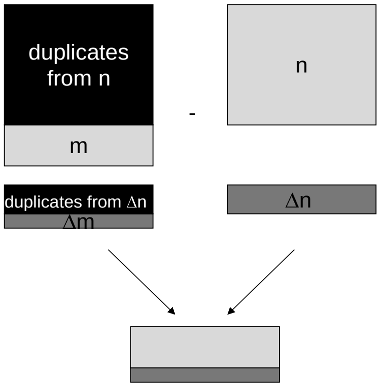


Figure 4.8: Incremental evaluation for set difference. Assume new data part Δm contains duplicates in Δn, but does not contain duplicates from the old part n. Then the results is $(m - n) + (\Delta m - \Delta n)$.

Chapter 5

Query Optimization

This chapter describes the query optimization techniques that are implemented in ARGUS, including transitivity inference, join ordering, conditional selection materialization, and minimum column projection.

The query optimization techniques help generate efficient shared query networks. Particularly, transitivity inference derives implicit highly-selective predicates from existing query predicates to filter out many non-result records in earlier stages and reduce the amount of data to be processed later. Join ordering aims to find the optimal join order to minimize the join intermediate result sizes. Conditional selection materialization decides whether the results of a set of selection predicates will be materialized or not based on selection factors. It aims to gain computation efficiency by balancing the computation saving and the materialization overhead. And minimum column projection projects the minimal set of columns for intermediate result tables shared by multiple queries to reduce intermediate result sizes.

Section 5.1 describes the transitivity inference in detail. Section 5.2 briefly overviews the minimum column projection and refers to Chapter 8 for implementation details. Sections 5.3 and 5.4 describe the problem and importance of conditional materialization and join order optimization. And Section 5.5 describes the design of the optimizer that performs conditional materialization and join order optimization.

5.1 Transitivity Inference

Transitivity inference explores the transitivity property of comparison operators, such as $>$, $<$, and $=$, to infer hidden selective selection predicates from a set of existing predicates. For example, in Example 1.1, the query has the following conditions (the first is very selective): $r1.amount > 1000000$, $r2.amount > r1.amount * 0.5$, and $r3.amount = r2.amount$. The first two predicates imply a new selective predicate on $r2$: $r2.amount > 500000$. Further, the third predicate and the newly derived predicate imply another new selective predicate on $r3$: $r3.amount > 500000$. These inferred predicates have significant impact on performance. The intermediate result tables of the highly-selective selection predicates are very small and save significant computation on subsequent joins.

Given a pair of predicates, if one condition contains exactly one column a, which we call single-table condition (STC), and the other contains exactly two columns a and b from two different tables, which we call joint-table conditions (JTC), then the inference module will try to infer some predicate on b.

The inference module is comprised of two parts. The first part builds up the data structures from the existing conditions. The second part loops through the data structures to look for the hidden conditions. Figure 5.1 shows the data structures and work flow of the inference module.

JTCIDHash is a hash table whose keys are JTC identifiers, and the values are JTCs. STCHash is a hash table that stores all the STCs whose keys are the columns, and the values are the lists of the STCs that contain the key columns. JTCHash is a hash table that stores all the JTCs whose keys are also column names, and the values are the lists of IDs of JTCs that contain the key columns. STCList is the list of STCHash keys.

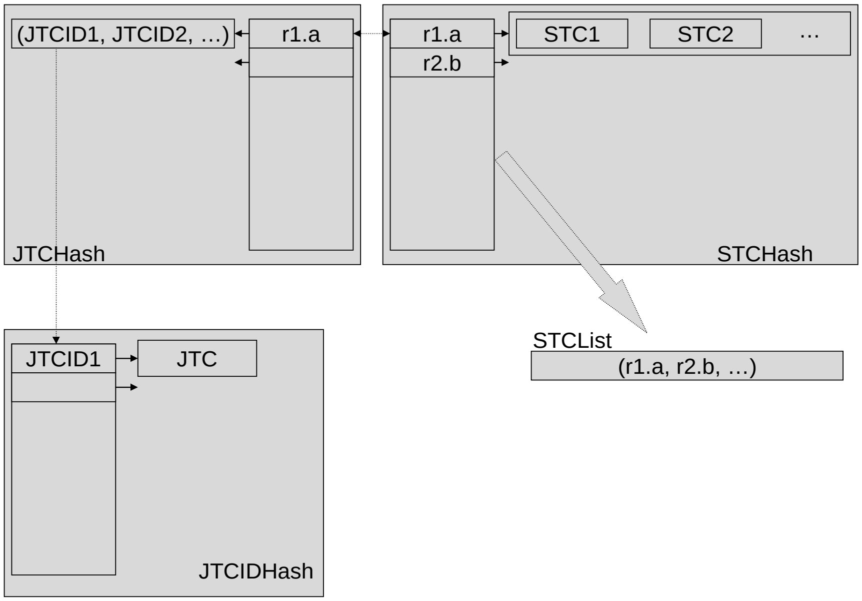


Figure 5.1: Data structures for transitivity inference. JTCHash hashes join conditions, which are actually stored in JTCIDHash. STCHash hashes selection conditions. A join condition in JTCHash and a selection condition in STCHash with the same hash keys are paired up for transitivity inference.

Following is the inference algorithm.

Algorithm 5.1 ***Inference***

1. *Pop up a column* $r1.a$ *from the STCList.*

2. *If there is an entry* $r1.a$ *in JTCHash, we do transitivity inference on each pair of the matching conditions: JTC1 and STC1.*

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

   – *Add* C *to the parsed subtree for the where-clause.*

   – *Add* C *to STCHash, and add* $r2.b$ *to STCList. This allows inferred conditions to be used for further inference.*

   – *Remove JTC1 from JTCHash. This step avoids endless cycle of inferring the same conditions. If we can't find the entry for a given JTC ID in JTCHash, then we simply skip this JTC.*

3. *Repeat Step 1 and 2 until STCList is empty.*

A transform function is developed to facilitate inference. Given a pair of STC and JTC, the function transforms the conditions into the following format:

$$\text{STC:} \quad r1.a >=< f_1(c)$$

$$\text{JTC:} \quad r1.a >=< f_2(r2.b)$$

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

The transitivity inference module is a part of the canonicalizer and it runs after the parsing.

Transitivity inference leads to up to 20-fold performance improvement in our experiments, shown in Section 10.2.

5.2 Minimum Column Projection

Minimum column projection refers to projecting the minimal set of columns for intermediate tables. When a new node is created, to save materialization space and execution time, we only project the necessary columns from its parents. These columns include those in the final results and those needed for further evaluation. The process becomes intricate when sharing is considered. When a node is shared, it may not contain all the columns needed for the new query. Then extra columns will be added to the node and possibly to its ancestors. This process is called *projection enrichment*. Aurora has the same functionality to project minimum columns. However, with its procedural query language, the sharing-related intricacy is not considered by the system but is handled manually. See Chapter 8 for details on projection management.

5.3 Conditional Materialization

In the incremental evaluation of selections/joins, materialized intermediate results improve performance by avoiding repetitive computations over the historical data. However, a potential problem is that when any materialized intermediate table is very large, thus requiring many I/O operations, the performance degrades severely. When intermediate results are not reduced substantially from the original data, the time saved from the repetitive computations may be offset or exceeded by the materialization overhead (I/O time).

Assume transitivity inference is not applicable by turning the module off, Example 1.1 is such a query. The two selection predicates ($r2.type\_code = 1000$, $r3.type\_code = 1000$) are highly non-selective, the sizes of the intermediate results are close to that of the original data table. Aware of the table statistics, such a materialization can be conditionally skipped, which we call *conditional materialization*.

Conditional materialization examines the selectivity of selection PredSets and decides whether or not materializing the PredSets based on a threshold cutoff (default is 0.3). Conditional materialization is implemented in the query optimizer.

PredSet selectivity may change over time. Dynamically detecting the change and modifying the materialization choice belong to the area of adaptive processing and will be future work.

Conditional materialization shows up to 1.8-fold performance improvement in our experiments, shown in Section 10.2.

5.4 Join Order Optimization

The optimal join order may lead to hundreds-fold performance improvement, as shown in classic query optimization literature [92, 106]. We implemented the join order optimization in the query optimizer.

5.5 Query Optimizer Design

Query optimization based on cost models has been well studied since the seminal paper [106]. [67] studied how to apply cost models to Gator networks. Similar techniques can be applied to ARGUS query network optimization. However, ARGUS has different optimization choices. It considers two problems, deciding whether or not materializing intermediate results, and optimizing the join order. Traditional query optimization concerns the join order, and choices of access methods, such as choosing using index or not, and choosing between merge join or nested-loop join, etc.

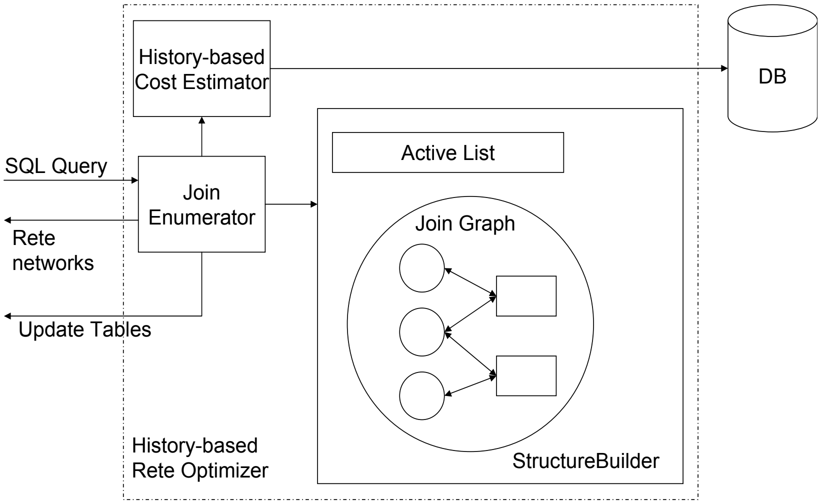


Figure 5.2: Query optimizer architecture. Join Enumerator enumerates sub-join-plans and chooses the cheapest one based on the cost estimates.

Figure 5.2 shows the architecture of the query optimizer. The join graph is a bipartite graph that shows the join connections of a query. The circle nodes represent the tables referenced in the query. Each circle node also represents the subplan to access the table. The rectangle nodes are join predicates whose associated edges link to the tables to be joined. The join graph is initialized by StructureBuilder. Initially, each subplan node is associated to a single table and contains the decision of conditional materialization. During the join-order optimization phase, the join graph is expanded to incorporate the new subplans.

For example, assume a query references four tables, table 1, 2, 3, and 4, and there are four sets of join predicates, denoted as $P(1,2)$, $P(2,3)$, $P(1,3)$, and $P(3,4)$, respectively. Particularly, $P(n,m)$ presents the set of join predicates that join tables n and m. The initial join graph is shown in Figure 5.3. Assume during the optimization, a subplan that joins tables 1 and 2 is generated, then the join graph is expanded to the graph shown in Figure 5.4. A new subplan node $(1,2)$ is generated, which can be seen as a wrap-up node of circle node 1, circle node 2, and the predicate node $P(1,2)$. The new node is connected to all the predicate sets whose join table sets have non-empty intersections with $\{1,2\}$.

Activelist is the list of active subplan nodes. An active subplan node will eventually be popped from the Activelist, and be used to search for expansion. Newly generated subplan nodes are put into the Activelist for further expansion until it is a complete plan. The position that the newly generated subplan is placed in the ActiveList is determined by the search strategy. When placing new subplans at the top of the list, where nodes are popped, it is the depth-first search. When placing new subplans at the bottom of the list, it is the breadth-first search. When the subplans in the Activelist maintain the order of estimated costs of the subplans, it is the hill-climbing search. And when the subplans maintain the order of a heuristic function, which is the sum of the estimated cost of the subplan and a non-overestimated cost of the remaining plan, then it is the A* search. The initial Activelist contains and only contains all the subplan nodes of the initial join graph.

StructureBuilder is the module that initializes and maintains the data structures in-

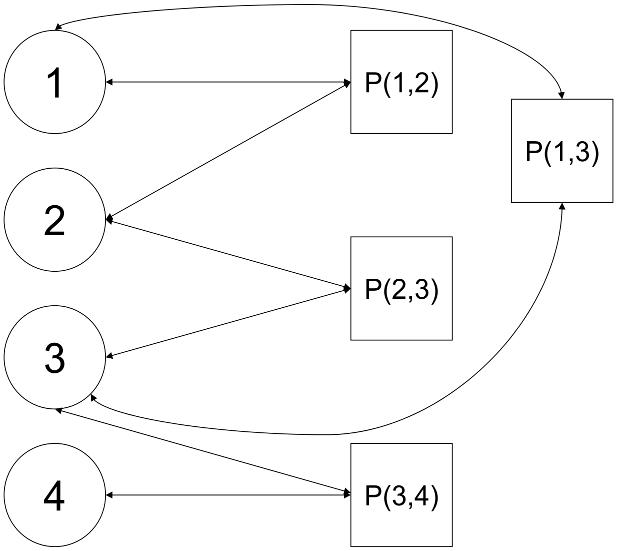


Figure 5.3: An initial Join Graph

cluding the join graph and Activelist.

JoinEnumerator is the module that searches the optimal construction plan. The search is driven by the Activelist. Given an active subplan PLAN1, JoinEnumerator generates a set of subplans that expand PLAN1. The cost of each plan is estimated by the history-based cost estimator. The newly generated subplans are checked for completeness, and if not, are put into the Activelist. When the Activelist is empty, the best complete plan is chosen as the output plan.

The history-based cost estimator estimates the cost of a query plan. It builds a temporary sub-query network in the engine environment, and runs the network against the existing data. The temporary sub-query network is created and expanded as the plan expands. Each subplan in the join graph has a correspondent network node whose associated table contains the output of the subplan. The estimator uses the execution time of the temporary networks on the existing data as the estimated cost of the networks running

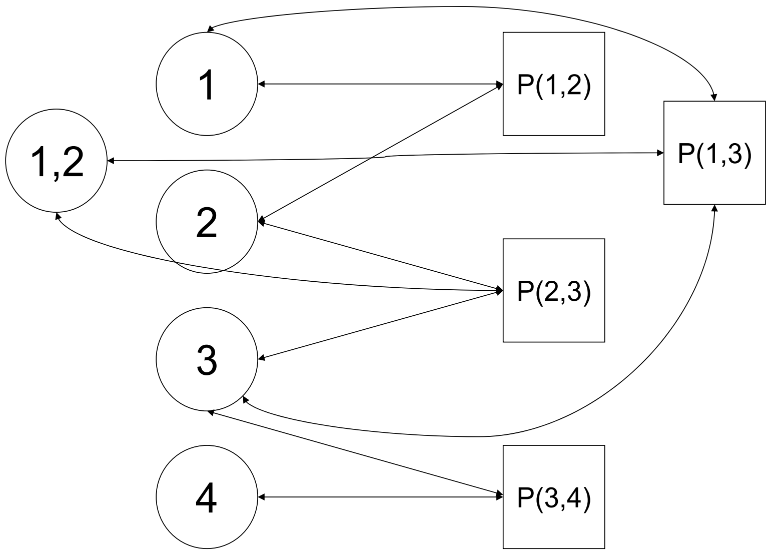


Figure 5.4: Expanding the Join Graph. The new node is the result of joinging nodes 1 and 2.

on the future stream data. The assumption adopted by the history-based cost estimator is that the future stream data distributions are the same to that of the historical data. It bypasses detailed modeling, and provides the accurate cost estimates given the same distribution assumption.

We also apply heuristic pruning to the optimization search. If applied heuristic functions predict an extremely high cost of a subplan, such subplan will not be expanded and estimated.

Chapter 6

Incremental Multiple Query Optimization on Selection and Join

This chapter describes the incremental multiple query optimization (IMQO) on selection and join queries. The next chapter describes the IMQO on aggregate and set operator queries. As discussed in Chapters 1 and 3, IMQO contains four complex steps: indexing the existing query network R, searching the common computations between the new query Q and the existing query network R, selecting the optimal sharing paths, and expanding the existing query network R to evaluate the new query as well. We focus on the first three steps in this chapter and the next chapter, and defer the fourth step to Chapter 9.

IMQO on selection and join queries is much more complex than that on aggregate or set operator queries due to rich query syntax and semantics. The query computation can be captured by a 4-layer hierarchical model, including layers of literal predicates, OR predicates (disjuncts of literal predicates, or ORPreds), predicate sets (conjuncts of ORPreds, or PredSets), and topology.

Section 6.1 introduces formal definitions on equivalent and subsumed literal predicates, ORPreds, and PredSets, and discuss PredSets operations. A reader may skip this section without losing the integrity on understanding the system. Section 6.2 presents the computation hierarchical model, related computation representation issues and solutions, and the common computation identification algorithms. Lastly Section 6.3 presents two sharing selection strategies, match-plan and sharing-selection.

6.1 Definitions

6.1.1 Equivalent Predicates

We say a predicate p_1 is equivalent to a predicate p_2, if for any database status D, p_1 and p_2 have the same results. An *equivalent predicate class* is the set of all equivalent predicates. And canonical predicate form is a representation of an equivalent predicate class.

Definition 6.1 (Equivalent Predicate) *For any database status D, if $p_1(D) = p_2(D)$, then p_1 is equivalent to p_2, and denote $p_1 \equiv p_2$.*

Definition 6.2 (Equivalent Predicate Class) *An equivalent predicate class C is a set of predicates, such that if $p_C \in C$, then*

- *$\forall p'_C$ that $p_C \equiv p'_C$, we have $p'_C \in C$.*
- *and $\forall p'_C$ that $\neg(p_C \equiv p'_C)$, we have $p'_C \notin C$.*

Definition 6.3 (Canonical Predicate Form) *A canonical predicate form F_C of an equivalent predicate class C is a representation of that class that satisfies the following conditions:*

- *Each equivalent predicate class has a unique canonical predicate form. The canonical predicate form can be viewed as the ID of an equivalent predicate class.*
- *Through legal transformations, such as mathematical transformations and rule-based inference, any predicate can be converted into the canonical predicate form.*

- *Canonical predicate form should be easy to be broken down to basic units such that different equivalent predicate classes can be compared and merged. For example, predicates $p_1 : t.a > 1$ and $p_2 : t.a > 0$ present two different equivalent predicate classes. From the semantics, we know that if p_1 is true, then p_2 is true, too. The inference on this kind of single-attribute predicates is made easy if the representation of the predicates comprises three separate parts: attribute name, comparison operator, and the compared constant.*

There are various canonical predicate forms. Our discussion is applicable to any canonical predicate form, but we assume that a system will stick to some particular canonical predicate form for consistency. The canonical predicate form is important to predicate indexing.

Because of the complexity of mathematical transformations and rule-based inference, the predicate indexing module of a system may not implement all transformations and inference. Therefore, the system may not recognize all forms of equivalent predicates. In this case, an equivalent predicate class is partitioned into small pieces. Each piece is a sub-equivalent predicate class. Therefore, there are cases that common predicates can be shared, but will not be shared because the common predicates are not identified by the system. Such a predicate indexing module is called *incomplete*.

Note that a predicate indexing module must be valid or correct. Validity means that the module never puts inequivalent predicates into one single equivalent predicate class, although it is allowed to fail to put equivalent ones into one class.

We can measure the quality of a predicate indexing module by its capability of identifying equivalent predicates. An incomplete predicate indexing module partitions an equivalent predicate class into smaller sub-equivalent predicate classes. Conceptually, we can use the number of sub-classes n_C, or the average number of sub-classes $n_{avg}(C)$, and the variance of n_C or variance of $n_{avg}(C)$, to measure the quality of the predicate indexing module. The complete predicate indexing module has $n_C = 1$, and $VAR_{n_C} = 0$. Although ARGUS canonicalization is not complete, it provides a much more close-to-complete identification capability than previous approaches by converting many syntactically-different predicates into the same canonical form.

6.1.2 Extending Predicate Set Operations

The extended predicate set operators are defined on predicate sets. We point out the distinction between the notation of a predicate set and the notation of the result tuple set obtained by applying the predicate set on the database D. A predicate set is a set of conjunctive predicates. For example, assume that $P_{NT} = \{p_1\}$ and $P_{OJ} = \{p_1, p_2\}$. P_{NT}, $\{p_1\}$, P_{OJ}, and $\{p_1, p_2\}$ are all predicate sets. The result tuple set obtained by applying a predicate set $\{p_1, p_2, ..., p_n\}$ is denoted as $S(\{p_1, p_2, ..., p_n\})$, and is a set of tuples in the database D (could be compound tuples generated by joining multiple tuples) that satisfy all the predicates in $\{p_1, p_2, ..., p_n\}$.

Fixing the database status as D, a predicate set P can be mapped to $S_D(P)$:

$S_D : P \rightarrow S_D(P)$

Vice versa is not necessarily true because there may be non-equivalent predicate sets P_1 and P_2 (see Definition 6.6 for equivalent predicate sets), such that $S_D(P_1) = S_D(P_2)$ for some particular database status D.

Among the definitions introduced below, the literal belong operator $\in$ has the similar meaning of the set operator $\in$ under the usual sense. Other set operators incorporate subsumption semantics.

Definition 6.4 (Literal Belong Operator $\in$) *Given a predicate p and a predicate set $P = \{p_1, p_2, ..., p_n\}$, if there is a p_k, k is an integer in $[1..n]$, such that $p \equiv p_k$, then we say that p literally belongs to P, and denote as $p \in P$.*

Definition 6.5 (Semantic Belong Operator $\in_\equiv$) *Given a predicate p and a predicate* *set P, if for any database status D, $S_D(P) \subseteq S_D(\{p\})$, then we say that p semantically belongs to P, and denote as $p \in_{\equiv} P$.*

If $p \in P$, then $p \in_{\equiv} P$. Vice versa is not true. For example, let $p_1 : t1.a > 1$, and $p_2 : t1.a > 2$, then $p_1 \notin \{p_2\}$, but $p_1 \in_{\equiv} \{p_2\}$.

Definition 6.6 (Equivalent Predicate Sets $\equiv_{\equiv}$) *Given two predicate sets P_1 and P_2, we say that P_1 and P_2 are equivalent, and denote as $P_1 \equiv_{\equiv} P_2$, if for any database status D, we have $S_D(P_1) = S_D(P_2)$.*

Definition 6.7 (Semantic Subset $\subseteq_{\equiv}$) *Given two predicate sets P_1 and P_2, we say that P_1 is a semantic subset of P_2, or that P_1 subsumes P_2, and denote as $P_1 \subseteq_{\equiv} P_2$, if for any database status D, we have $S_D(P_1) \supseteq S_D(P_2)$.*

Definition 6.8 (Proper Semantic Subset $\subset_{\equiv}$) *Given two predicate sets P_1 and P_2, we say that P_1 is a proper semantic subset of P_2, and denote as $P_1 \subset_{\equiv} P_2$, if $P_1 \subseteq_{\equiv} P_2$ and P_1 is not equivalent to P_2.*

Definition 6.9 (Semantic Superset $\supseteq_{\equiv}$) *Given two predicate sets P_1 and P_2, we say that P_1 is a semantic superset of P_2, or that P_1 is subsumed by P_2, and denote as $P_1 \supseteq_{\equiv} P_2$, if $P_2 \subseteq_{\equiv} P_1$.*

Semantic superset and semantic subset are also referred as query containment [49] in literature.

Definition 6.10 (Proper Semantic Superset $\supset_{\equiv}$) *Given two predicate sets P_1 and P_2, we say that P_1 is a proper semantic superset of P_2, and denote as $P_1 \supset_{\equiv} P_2$, if $P_2 \subset_{\equiv} P_1$.*

Consider two semantic subset examples. In the first example, let $P_1 = \{p_1\}$, and $P_2 = \{p_1, p_2\}$, then any tuple satisfying predicate set P_2 must also satisfy predicate set P_1, therefore $P_1 \subseteq_{\equiv} P_2$, which is straightforward as P_2 literally contains all the predicates in P_1.

In the second example, let $p_1 : t1.a > 1$, $p_2 : t1.a > 2$, $P_1 = \{p_1\}$, and $P_2 = \{p_2\}$. Similarly, any tuple satisfying predicate set P_2 must also satisfy predicate set P_1, therefore $S_D(P_1) \supseteq S_D(P_2)$. Under the semantic subset definition, we have $P_1 \subseteq_{\equiv} P_2$. Recall that $P_2' = \{p_1, p_2\} \equiv P_2$, it becomes clear that $S_D(P_1) \supseteq S_D(P_2')$.

The process to find the equivalent predicate set P_2' for the purpose of establishing the semantic relationship (or subsumption) between predicate sets is called *explication*, and say that P_2' is the *explication* of P_2 on P_1.

Definition 6.11 (Explication) *Given two predicate sets P_1 and P_2, $P_1 = \{p_{11}, p_{12}, ..., p_{1n_1}\}$, and $P_2 = \{p_{21}, p_{22}, ..., p_{2n_2}\}$. We say predicate set $P_2' = \{p_{31}, p_{32}, ..., p_{3n_3}\}$ is the explication of P_2 on P_1, denote as $P_2' = E_{P_1}(P_2)$, if all the following conditions are satisfied.*

- *$\forall p_{2k} \in P_2$, we have $p_{2k} \in P_2'$.*
- *$\forall p_{1k} \in P_1$, if $p_{1k} \in_{\equiv} P_2$, then we have $p_{1k} \in P_2'$.*
- *$P_2 \equiv_{\equiv} P_2'$*

We can see that in terms of literal predicate elements, P_2' is a superset of P_2.

Definition 6.12 (Semantic Intersection $\cap_{\equiv}$) *Given two predicate sets P_1 and P_2. Let $P_1' = E_{P_2}(P_1)$, and $P_2' = E_{P_1}(P_2)$. The semantic intersection P of P_1 and P_2, denoted as $P = P_1 \cap_{\equiv} P_2$, is defined as a predicate set such that*

- *$\forall p \in P$, we have $p \in P_1'$, and $p \in P_2'$.*
- *$\forall p$, such that $p \in P_1'$, and $p \in P_2'$, then we have $p \in P$.*

For example, let p_1, p_2, and p_3 be three selection predicates on three different columns of a table respectively, such as $p_1 : t1.a = 1$, $p_2 : t1.b = 2$, and $p_3 : t1.c = 3$. Let $P_1 = \{p_1, p_2\}$, and $P_2 = \{p_1, p_3\}$, then we have $P_1 \cap_{\equiv} P_2 = \{p_1\}$.

In another example, let $p_1 : t1.a > 1$, $p_2 : t1.a < 3$, $p_3 : t1.a > 2$, $p_4 : t1.a < 4$, $P_1 = \{p_1, p_2\}$, and $P_2 = \{p_3, p_4\}$. Then we have $E_{P_2}(P_1) = \{p_1, p_2, p_4\}$, and $E_{P_1}(P_2) =$

$\{p_3, p_4, p_1\}$. Therefore, we have $P_1 \cap_{\equiv} P_2 = \{p_1, p_4\}$. The explication process is important here for calculating the intersection.

We shall note that $S(P_1 \cap_{\equiv} P_2) \supseteq S(P_1) \cup S(P_2)$.

Since ARGUS does not yet support restructuring, it does not need to compute the predicate set intersections and thus does not implement the explication yet. Explication, intersections, and restructuring will be supported in future. See Section 11.2.1 for the future work.

6.2 Indexing and Searching

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

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

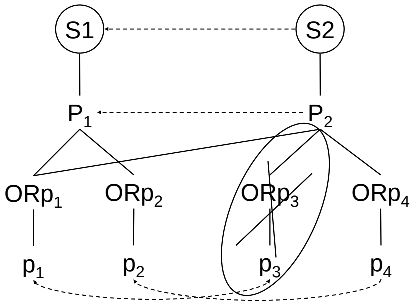


Figure 6.1: Computation hierarchy.

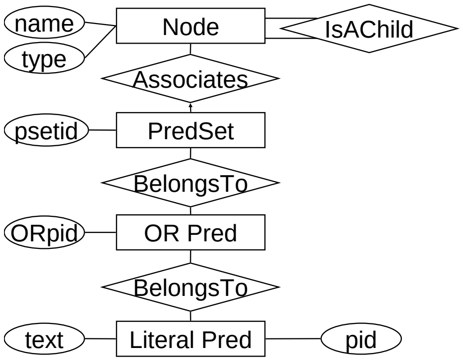


Figure 6.2: Hierarchy ER model.

Figure 6.1 shows the hierarchy for the two nodes $S1$ and $S2$ from Figure 1.1. 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 6.2.

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

There are several issues we need to consider before we transform the 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.

6.2.1 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 as done by previous work. For doing so, we introduce a canonicalization procedure. It transforms syntactically-different yet semantically-equivalent literal predicates into the same pre-defined canonical form. Then the equivalence can be detected by exact string match.

There is intricacy with regard to the canonicalization. We need to identify subsumption relationship between literal predicates. For example, $t1.a > 10$ subsumes $t1.a \geq 5$. The exact match on the canonicalized predicates can not identify subsumptions. Instead, the subsumption can be identified by a combination of the exact match on the column references, the operator comparison, and the constant comparison. 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 extremely rich syntax and unknown semantics, e.g. user-defined functions, a complete canonicalization procedure is impossible. Previous works [56, 34, 84] apply simple approaches to identify subsumptions between simple selection predicates (e.g. $t1.a > 10$ subsumes $t1.a > 5$), equivalence of equi-join 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, the 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 an one-time operation for just new queries, and the average predicate length is far less than the extent that can slow 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.

Algorithm 6.1 ***(Predicate Canonicalization)***

- *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.

Algorithm 6.2 ***(Expression Canonicalization)***

- *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 leave the function call as it is.*
- *Flattening and sorting commutable sub-expression parse 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.* $a * (b + c) \Rightarrow a * b + a * c$.

6.2.2 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 ρ_1: $F.c = 1000$, and the other a self-join predicate ρ_2: $T1.a = T2.b$. The canonicalized ρ_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 ρ_2. Therefore, ρ_1's STA, $T1$ or $T2$, must be indexed in p_1. Similar situation exists in PredSets where some ORPred is a selection from a single table and some other is a self-join. 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[1] 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.

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

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.

We note that other work, such as STREAM, fails to support self-joins for querying streams, requiring very inefficient data replication.

6.2.3 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. See Section 11.2.1 for the future work on re-optimization.

Identifying subsumptions between PredSets is NP-hard [75, 99]. 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 the acceptance and rejection can be computed in 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 [13]. Searching all functional dependencies is also NP-complete [98], 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.

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 subsumable relationship between pairs of operators based on the order of the right sides.

Definition 6.13 *For two literal operators γ_1 and γ_2 and an order O, we say (γ_1, γ_2, O) is a subsumable triple if following is true: for any pair of canonicalized literal predicates ρ_1 and ρ_2, assume ρ_1: $L(\rho_1)\gamma_1 R(\rho_1)$, and ρ_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 $\rho_1 \rightarrow \rho_2$.*

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 6.1 shows the implemented subsumable triples. With this, look-up queries can be formulated to retrieve the indexed subsumption literals in constant time.

γ_1	γ_2	Order O		γ_1	γ_2	Order O
$>$	$>=$	E		$<$	$<=$	E
$=$	$>=$	E		$=$	$<=$	E
$>$	$>=$	D		$>$	$>$	D
$>=$	$>=$	D		$>=$	$>$	D
$=$	$>$	D		$=$	$>=$	D
$<$	$<=$	I		$<$	$<$	I
$<=$	$<=$	I		$<=$	$<$	I
$=$	$<$	I		$=$	$<=$	I

Table 6.1: Subsumable Triples (γ_1, γ_2, O). E is equal, D is decreasing, and I is increasing.

6.2.4 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 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 the PredSet layer. Identifying equivalence 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 6.3 shows that each of the l literals in p is subsumed by s indexed literals, and each indexed literal belongs to m different ORPreds.

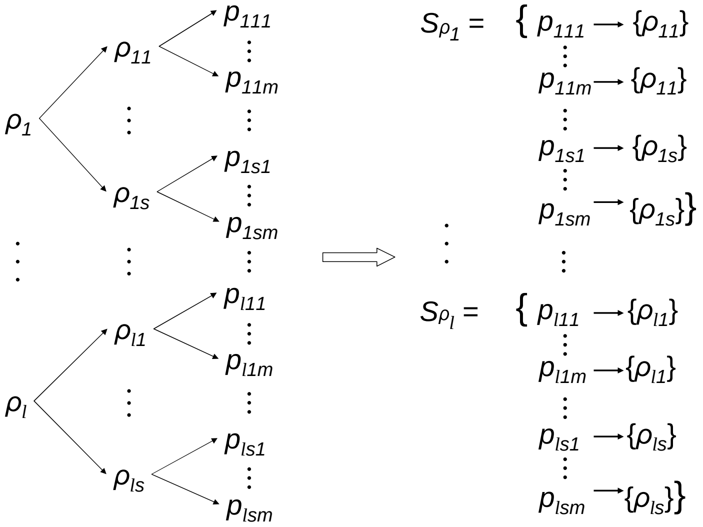


Figure 6.3: Subsumption and 2-level hash sets.

According to Section 6.2.3, 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 ρ_i, a subsumption exists between ρ_{ij} and ρ_{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, ρ_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 structure 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 6.3, for each literal ρ_i, the subsuming literal predicates and their ORPreds form a 2-level hash set S_{ρ_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 6.3, the bottom-level sets are singleton sets whose elements are literals that subsume p.

We define a binary operation Υ-intersection $\cap_\Upsilon$ 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 6.14 *Given two 2-level-hash sets S_1 and S_2, we say S is the Υ-intersection of S_1 and S_2, denoted as $S = S_1 \cap_\Upsilon S_2$, if and only if following is true: S is a 2-level-hash set, $keys(S) = keys(S_1) \cap keys(S_2)$, and for $\forall k \in keys(S)$, $S(k) = S_1(k) \cup S_2(k)$.*

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. Υ-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 6.3, the time of Υ-intersecting two hash sets is $O(s*m)$.

Algorithm 6.3 ***Subsumed_ORPreds***
input: *p, R;* ***output***: *SubsumedSet(p)*

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

```
$I := \cap\Upsilon_{i=1}^{l} S_{\rho_i}$;

$SubsumedSet(p) := \{\}$;
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_{ρ_i}, $i \in [1..l]$, Υ-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 by 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 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 by 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.

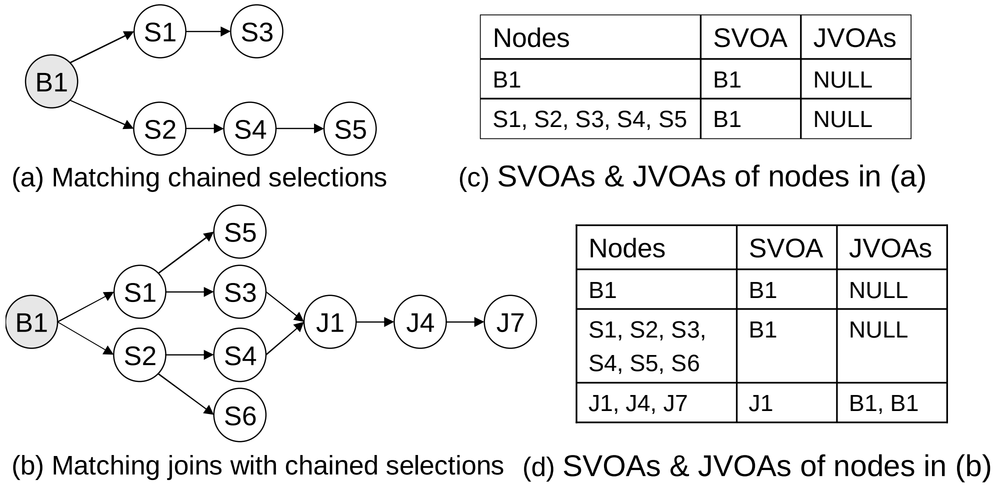
6.2.5 Topology Connections

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 types of ancestors. An important one is the DPredSet which connects N to its direct parents. DPredSet is used in constructing the execution code, and needs to be indexed.

Solely relying on DPredSets does not provide the fast searching. In Figure 6.4(a), assume a selection PredSet P from stream table $B1$ can be evaluated from any of $S1$, $S2$, $S3$, $S4$, and $S5$, and $S5$ 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 6.4(a) searching the descendants of $S1$ as well.

Same problems exist for join PredSets. In Figure 6.4(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



(a) Matching chained selections

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

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

(b) Matching joins with chained selections

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

Nodes	SVOA	JVOAs
B1	B1	NULL
S1, S2, S3, S4, S5, S6	B1	NULL
J1, J4, J7	J1	B1, B1

Figure 6.4: Multiple topology connections.

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 6.4(c) and 6.4(d) show the SVOAs and JVOAs for nodes in (a) and (b).

Definition 6.15 (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 6.16 (JVOA) *A join node N's JVOAs are the closest ancestor nodes that are either join nodes (but not N) or base stream nodes. A selection node N's JVOAs are the JVOAs of N's SVOA. And a base stream node's JVOA is NULL. JVOA stands for join very original ancestor.*

SVOAs present local selection chains, and JVOAs present one join depth beyond the local selection chains. With SVOAs and JVOAs, the chained searches are no longer necessary. 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 6.4(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.

6.2.6 Predicate and PredSet Conversions

When a new node is created or the topology of a sub-network is changed, the topology information in the system catalog has to be updated to maintain the consistent state with the changed topology. To do that, we identify all the nodes whose topology information needs to be updated, compute the updated information, and make the corresponding changes to the system catalog. When a new node is created, only that node's topology information needs to be added into the system catalog. When any existing node is moved around, then we need to decide which descendants of it need to be updated and what information (JVOA, SVOA, and/or Direct Parent(s)) needs to be updated. For each node, the needed information to be updated is computed according to its predicate sets from the direct parent(s), JVOAs, and SVOA.

The computation usually applies the combination of the predicate set conversion, predicate set union, and predicate set difference to obtain the right predicate sets in the right forms.

PredSet conversion converts a PredSet from one form to another equivalent form. In a new query Q, the PredSet is presented in JVOA or SVOA form. In either form, the PredSet references the JVOA or SVOA tables and their columns. When a PredSet is indexed, we need other forms, such as direct form, in which the PredSet references the direct parent tables and their columns. The conversions between the different forms are performed in the procedures similar to query rewriting. The conversions replace the table and column references in one form by the table and column references in the other form.

PredSet union is a binary PredSet operator similar to set union but drops the duplicate subsumption predicates. Algorithm 6.4 shows the implemented PredSet union algorithm. We add each predicate $p \in P_1$ to P, unless p subsumes some $p' \in P_2$ and p' is not equivalent to p. Then we add each predicate $p' \in P_2$ to P, unless p' subsumes some $p \in P_1$. This assures that no predicate in P subsumes another in P, and that P is equivalent to the union of P_1 and P_2.

Algorithm 6.4 *PredSet Union*
input: P_1, P_2
output: $P = P_1 \cup P_2$

```
$P := \phi$
for each predicate $p \in P_1$
    if there is a $p' \in P_2$, such that $p' \rightarrow p$ and $\not\exists(p' \equiv p)$
        next;
    $P+ := p$

for each predicate $p' \in P_2$
    if there is a $p \in P_1$, such that $p \rightarrow p'$
        next;
    $P+ := p'$
```

PredSet difference is a binary PredSet operator similar to set difference, as shown below.

Algorithm 6.5 ***PredSet Difference***
input: P_1, P_2
output: $P = P_1 - P_2$

```
$P := \phi$
for each predicate $p \in P_1$
    if there is a $p' \in P_2$, such that $p' \to p$
        next;
    $P+ := p$
```

6.2.7 Relational Model for Indexing

Now we consider converting the indexing ER model to the relational model.[2] Two adjustments are made. First, the relations that index literal predicates and ORPreds are merged into one, *PredIndex*, based on the assumption that ORPred are not frequent in queries. This allows a literal predicate to appear multiple times in *PredIndex* if it belongs to different ORPreds. But this redundancy is negligible given the assumption. The second adjustment is splitting the node topology indexing relation (Node Entity in the ER model) to two, namely, *SelectionTopology*, and *JoinTopology*, based on the observation that the topology connections on selection nodes and on join nodes are quite different.

Figure 6.5 shows the indexing relation schemas. In *PredIndex*, 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* record 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

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

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 6.5: System Catalog Schemas

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 6.2.5. *JoinTopology* does not need to index SVOA.

Section 10.4 evaluates the effects of IMQO sharing supported by the indexing and searching scheme. It shows up to hundred-fold performance improvement over non-sharing approaches. The section also shows the effectiveness of canonicalization which leads to tens-fold performance improvement.

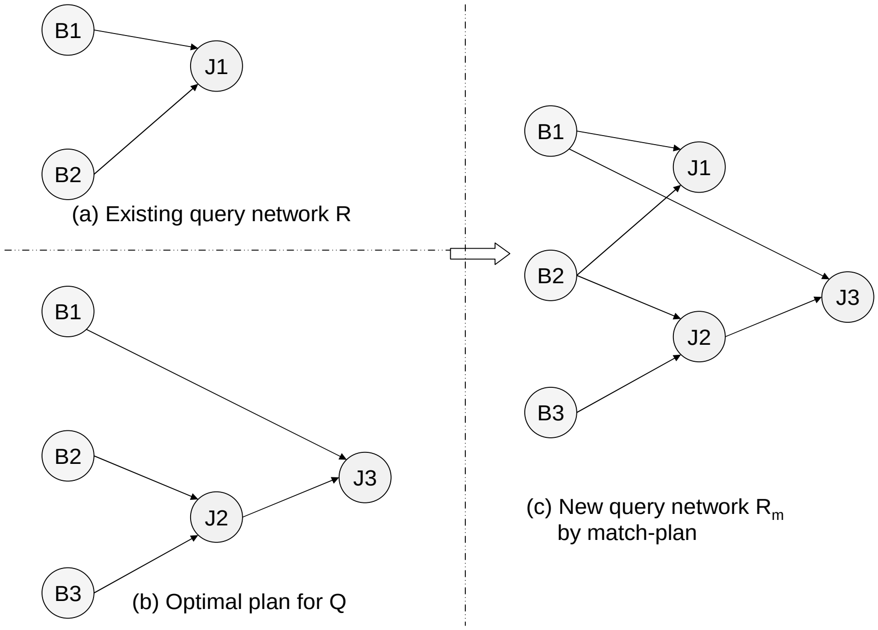


Figure 6.6: Match-Plan matches the pre-optimized plan structures with the existing query network.

6.3 Sharing Strategies

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

Figures 6.6 & 6.7 illustrate the difference between sharing-selection and match-plan. Assume the existing query network R (Figures 6.6(a) & 6.7(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 6.6(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,

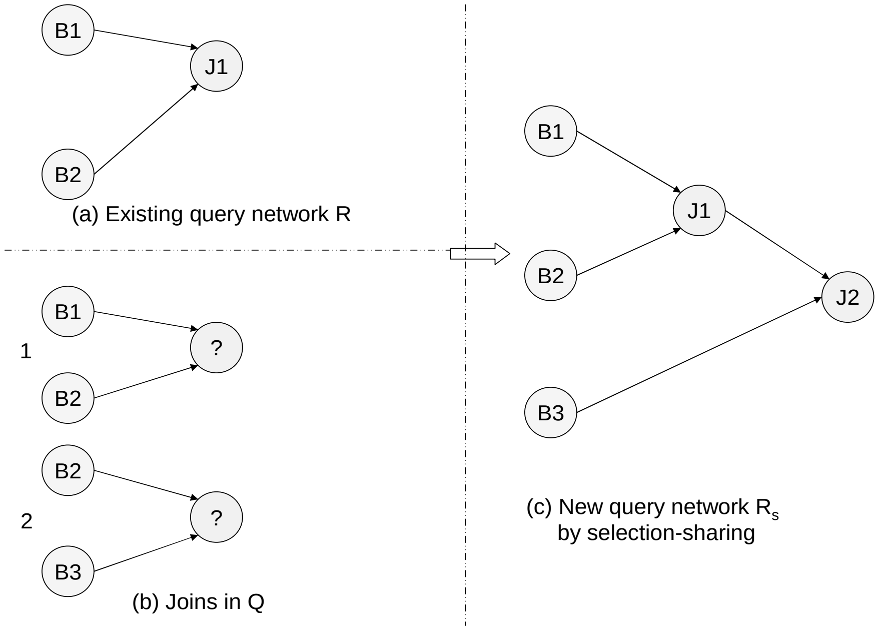


Figure 6.7: Sharing-Selection selects the sharable joins and expands the existing query network with unsharable ones.

thus no sharing is available. It expands R to a new query network R_m (Figure 6.6(c)). From sharing-selection, both of the joins (Figure 6.7(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 6.7(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.

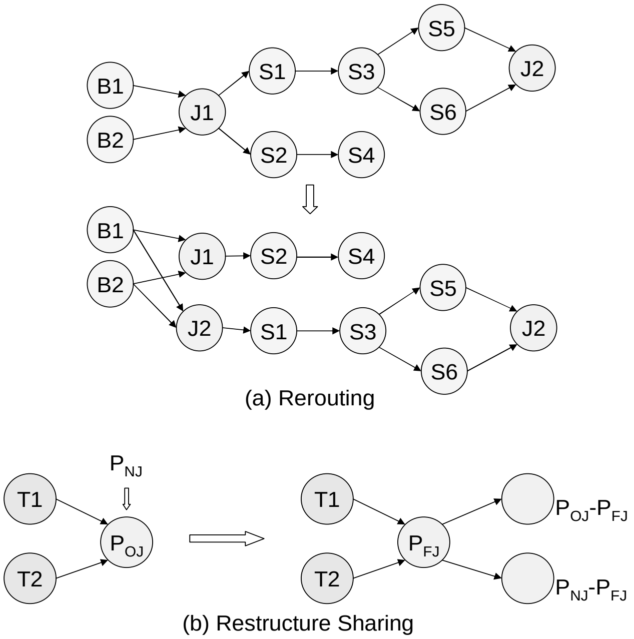


Figure 6.8: Complex Sharing. Rerouting reconnects a local subtree to a newly created node $J2$. Restructuring changes the join node computation so it can be shared by more nodes.

In both strategies, we need to choose the optimal sharable node at each JoinLevel. We apply a simple cost model for doing so. The cost of sharing a node S is simply the cost of evaluating the remaining part of the chosen PredSet P. The cost is defined as the size of S, the number of records to be processed to obtain the final results of P. For example, in Figure 4(a), assume P can be evaluated from $S5$. 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.

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 1.5. This choice is applied by both sharing-selection and match-plan.

The two sharing strategies are compared in Section 10.4, and sharing-selection is shown to be better than match-plan in general.

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 6.8(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 re-optimizes 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 6.8(b). The choices to perform these topological operations should be decided by cost models, and are also applicable to 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 re-optimization.

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.

Chapter 7

Incremental Multiple Query Optimization on Aggregates and Set Operators

This chapter describes the incremental multiple query optimization (IMQO) on aggregates (Section 7.1) and set operators (Section 7.2). As mentioned before, the previous chapter describes the IMQO on selection and join queries. Both chapters focus on the first three steps of IMQO, indexing, searching, and selecting. And the last IMQO step, expanding, is described in Chapter 9.

7.1 Incremental Multiple Query Optimization on Aggregates

As discussed in Section 4.2, algebraic aggregate functions, such as MIN, MAX, COUNT, SUM, AVERAGE, STDDEV, and TrackClusterCenters, can be incrementally updated upon data changes without revisiting whole history of grouping elements; while holistic aggregate functions, e.g. quantiles, MODE, and RANK, can not be done this way.

The algebraic functions can also be shared through dimension reductions, but holistic functions can not.

Reconsider the query A described in Section 4.1 that monitors the number of visits and the average charging fees on each disease category in a hospital everyday. When new tuples from the stream Med arrive, the aggregates *COUNT(*)* and *AVERAGE(fee)* can be incrementally updated if *COUNT(*)* and *SUM(fee)* are stored, shown in Figure 4.3.

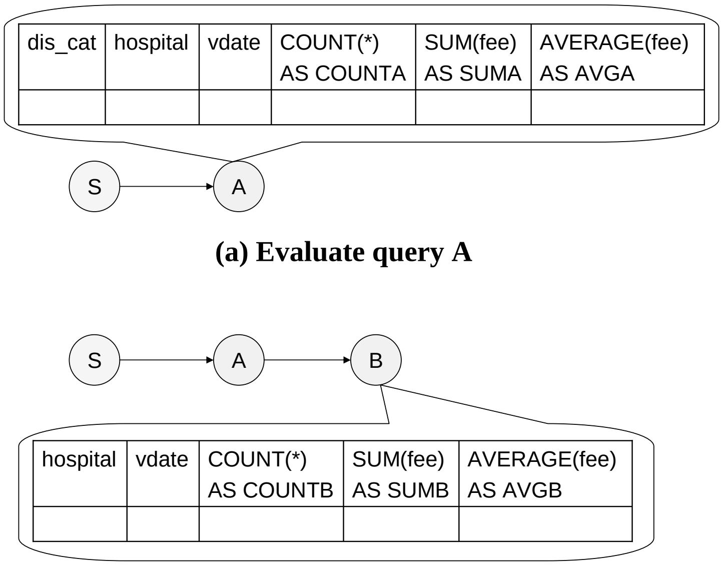
Now consider a new query B that monitors the number of visits and average charging fees in a hospital everyday, as shown in Example 7.1. B groups can be obtained by compressing A groups on the $CAT(disease)$ dimension. Further, B's aggregate can be obtained from A as well. Thus the system shares A's results to evaluate B, as shown in Figure 7.1(b). This sharing process is called *vertical expansion*.

Note that holistic functions can not be shared through vertical expansion since the aggregate values of B can not be obtained from the aggregate values of A.

Example 7.1 **(**B**)** *Monitoring the number of visits and the average charging fees in a hospital everyday.*

$$\begin{array}{ll}
SELECT & hospital, vdate, \\
 & AVERAGE(fee) \\
FROM & Med \\
GROUP\ BY & hospital, \\
 & DAY(visit\_date)\ AS\ vdate
\end{array}$$

Aggregate-related system catalog relations include *GroupExprIndex*, *GroupExprSet*, *GroupTopology*, *GroupColumnNameMap*. *GroupExprIndex* records all canonicalized group expressions. *GroupExprSet* records the dimension sets, and *GroupTopology* records the topological connections of the aggregate nodes. A dimension set is the set of groupby expressions in an aggregate query. The dimension set specifies the grouping criteria. Each dimension set has a *GroupID*, is uniquely associated with an aggregate node in *GroupTopology*, and has one or more GroupExpressions recorded in *GroupExprIndex*. Each node has a unique entry in *GroupTopology* which records the node name, its direct parent (the



dis_cat	hospital	vdate	COUNT(*) AS COUNTA	SUM(fee) AS SUMA	AVERAGE(fee) AS AVGA

(a) Evaluate query A

hospital	vdate	COUNT(*) AS COUNTB	SUM(fee) AS SUMB	AVERAGE(fee) AS AVGB

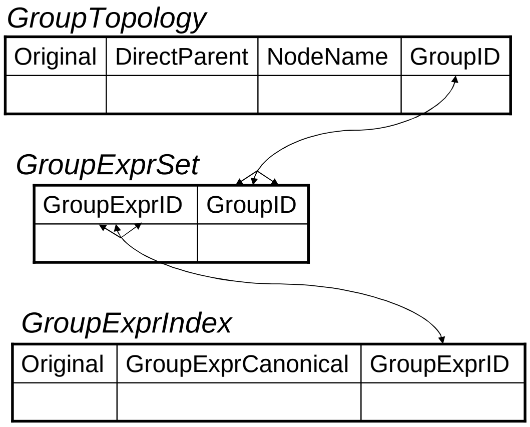
(b) Evaluate query B from A

Figure 7.1: Evaluating Queries A and B

node from which the results are computed), its original stream table, and the *GroupID*. *GroupColumnNameMap* records the projected group expressions and aggregate functions for each node; see Section 8.1.2.

Similar to the IMQO on selection and join queries, as shown in Figure 3.4, the IMQO on aggregate queries works as following for a new query B:

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



GroupTopology

Original	DirectParent	NodeName	GroupID

GroupExprSet

GroupExprID	GroupID

GroupExprIndex

Original	GroupExprCanonical	GroupExprID

Figure 7.2: Aggregate System Catalog

- Create a new node B by creating the table pairs and updating the system catalog.
- Perform rerouting on B.

7.1.1 Vertical Expansion and Sharing Strategies

Vertical expansion is not applicable to holistic queries. However, holistic queries can still be shared if they share the same dimension sets. Additional aggregate functions requested by the new query should be added to the holistic node. Such process is called *horizontal expansion*. In following discussion, we focus on sharing among distributive and algebraic functions.

Query B can be evaluated from the query A (see Figure 7.1). Figure 7.3 shows how the *vertical expansion* creates and initializes B from A. The process is comprised of two steps and takes the time of $T^{VInit} = O(|A_H|)$. The further-aggregate step executes the

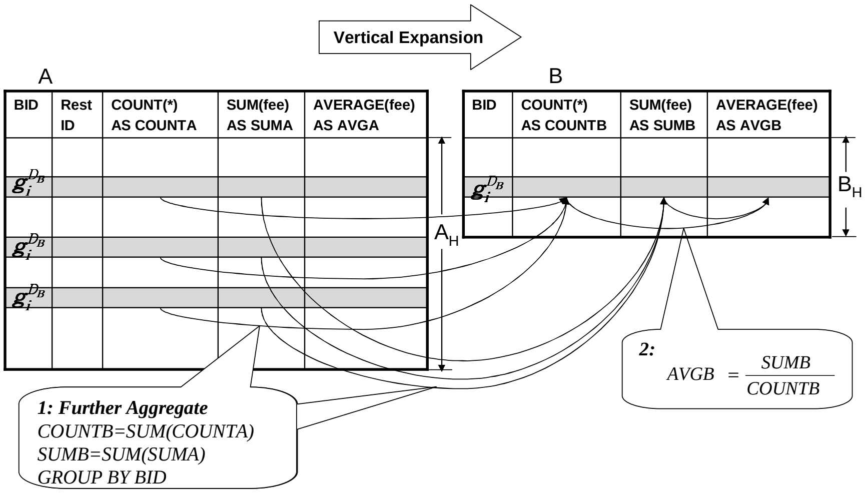


Figure 7.3: Vertical Expansion, IMQO for aggregate queries. This shows that query B is evaluated from query A's results.

code instantiated from the vertical expansion rules stored in the *AggreRules*, and the algebraic-computing step is applicable only to algebraic functions.

Figure 7.4 shows the incremental aggregation for $AVERAGE(fee)$ in query B. It is the same to the procedure shown in Figure 4.4 except the first step which performs further aggregation from A_N instead of from S_N. The time complexities are following: $T^V_{curr} = O(|A^2_N| + |B_H|)$, $T^V_{built-in} = O(|A_N| + |B_H|)$, and $T^V_{prefetch} = O(|A_N|)$.

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

After the new node B is created, the system invokes the rerouting procedure. It checks if any existing node C can be sped up by being evaluated from B. We apply a simple cost

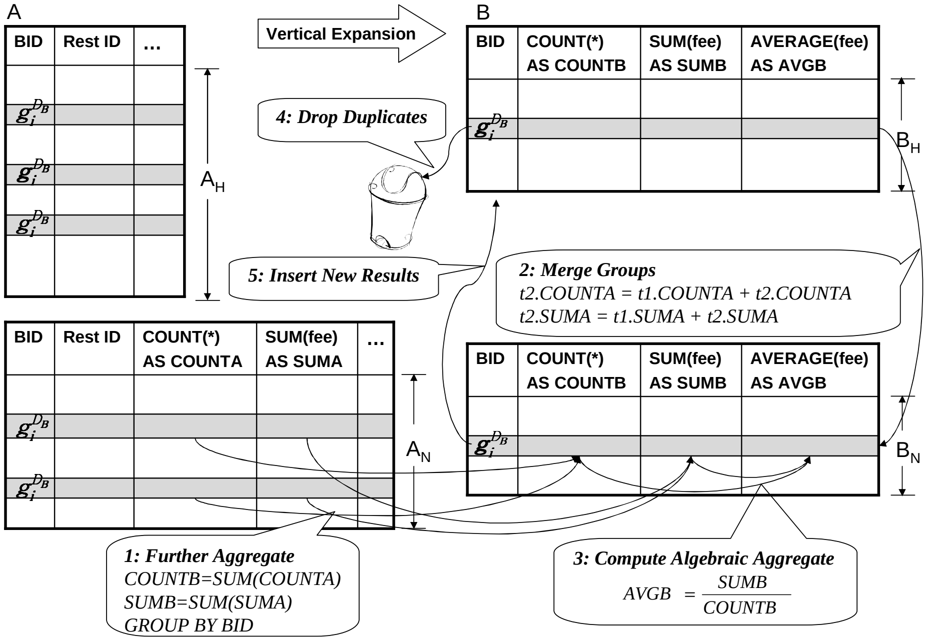


Figure 7.4: Incremental Aggregation on Vertical Expansion. This shows incremental aggregation can be realized on vertical-expanded aggregate node B as well.

model to decide such rerouting nodes. If 1. B contains all the aggregate functions needed by C, and 2. $|B_H| < |P_H^C|$ where P^C is C's current parent node, then C will be rerouted to B. The system applies a simple pruning heuristic. If a node C satisfies both conditions, and a set of nodes $\{C_i\}$ satisfying the first condition are descendants of C, then any node in $\{C_i\}$ should not be rerouted, and so are dropped from consideration.

Section 10.3 evaluates the effects of vertical expansion. It shows up to hundred-fold performance improvement over non-sharing approaches.

7.2 Incremental Multiple Query Optimization on Set Operators

The IMQO on set operators is similar to those on selection/join queries and aggregate queries. In this section, we describe the indexing and searching on set operator computations, and the sharing strategy to select the optimal sharable nodes.

7.2.1 Indexing on Set Operator Nodes

A set operator node is the result of a set operation on two or more nodes. A set operator node can be shared if it provides the data from which final results can be computed. To do this, we index topology of set operator nodes.. First we introduce **set operator node cluster** and **veryorig table set**.

Definition 7.1 (Set Operator Node Cluster) *A set operator node cluster is a maximum connected sub-graph of the query network. All the nodes in the sub-graph are set operator nodes. It is defined recursively as following:*

1 A set operator node cluster G contains at least one set operator node N.

2 If any of N's direct parent nodes, N_d, is a set operator node, N_d is in G.

3 If any of N's child nodes, N_c, is a set operator node, N_c is in G.

Definition 7.2 (Veryorig Table Set) *The veryorig table set $\{N_v\}$ of a set operator node N in the set operator node cluster G is defined as following:*

1 If any of N's direct parent node N_d is not in G, N_d is in $\{N_v\}$.

2 For any N's ancestor M that is in G, if any of M's direct parent node M_d is not in G, M_d is in $\{N_v\}$.

Theorem 7.1 *Any veryorig table of a set operator node is non-set-operator node.*

Therefore, although set operator nodes can construct multi-layer topology, conceptually, they are considered as being in just one single-layer starting from their non-set-operator veryorig tables. This allows the global search within set operator node clusters. See Figure 7.5.

Set operator nodes are indexed in three system catalog relations. *UnionTopology* indexes the topology of set operator node clusters, including the veryorig tables and the

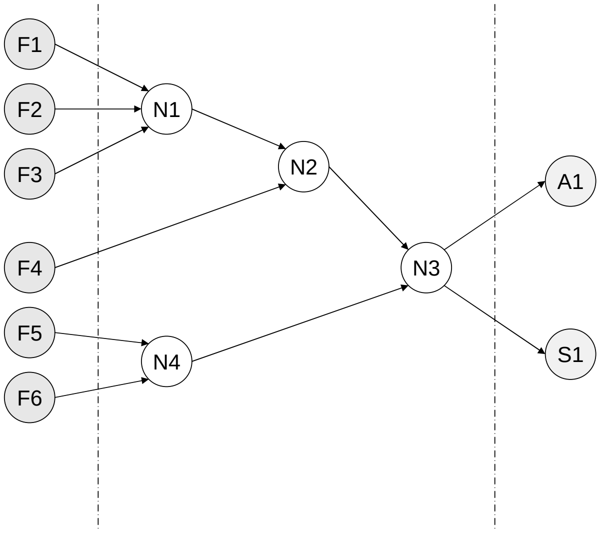


Figure 7.5: A set operator node cluster, N1-N4. F1-F6 are the veryorig non-set-operator nodes. A1 is an aggregate node and S1 is a selection node.

direct parents. *UnionNode* indexes node-specific information, such as set operators in its direct form and its veryorig form, and Boolean flags to control selective evaluation. And *UnionColumnNameMap* indexes column projection information; see Section 8.1.

ResultTable	Node N.
DirectUnionType	N's set operator from direct parent nodes.
VeryOrigUnionType	N's set operator from veryorig nodes.
ReteFlag	N's ReteFlag.

Figure 7.6: UnionNode: Set operator node indexing.

7.2.2 Searching Sharable Set Operator Nodes

Given a new set operator query Q that operates on a set of tables $\{M_v\}$, we want to find a set of set operator nodes $\{N\}$ from the existing query network R where N can be used

ResultTable	Node N.
DirectOrigTableName	N's direct parent N_d.
DirectOrigUnionSubID	N_d's position ID to evaluate N.
VeryOrigTableName	N's veryorig table N_v.
VeryOrigUnionSubID	N_v's position ID to evaluate N.

Figure 7.7: UnionTopology: Topology indexing for set operator nodes.

to evaluate Q. The sharability depends on the set operators, the veryorig tables, and the columns. First we state the sharability criteria on the set operators and the veryorig tables in Algorithm 7.1. Then we state the criteria on the columns in Algorithm 7.2.

Algorithm 7.1 ***Sharability on set operators and veryorig tables***

- *When Q is set difference (MINUS), a sharable N can be either of the following:*

  1. *N is set difference;*
  2. *first VeryOrigTables of N and Q are the same; and*
  3. *Q's remaining set of VeryOrigTable is a superset of N's remaining set of VeryOrigTable.*

  or

  1. *N is set union (UNION); and*
  2. *Q's remaining set of VeryOrigTable (excluding the first VeryOrigTable) is a superset of N's VeryOrigTable set.*

- *When Q is UNION ALL, a sharable N should satisfy:*

  1. *N is UNION ALL; and*
  2. *Q's VeryOrigTable set is a superset of N's VeryOrigTable set*

- *When Q is UNION, a sharable N should satisfy:*

  1. *N is either UNION or UNION ALL; and*
  2. *Q's VeryOrigTable set is a superset of N's VeryOrigTable set.*

For a sharable N, Q's column set must be a subset of N's column set, and their column position mapping across veryorig tables must be consistent. The consistency checking can be performed as following.

Algorithm 7.2 ***Column Position Check***

1. *For the first VeryOrigTable of Q and N, we record the matching columns' position mapping. For example, mapping column positions in Q (selecting 4 columns) to those in N (selecting 5 columns), we get* $(1, 2, 3, 4) \rightarrow (1, 2, 5, 3)$. *This means that the 1st column of Q matches the 1st of N, 2nd of Q matches the 2nd of N, 3rd of Q matches the 5th of N, and 4th of Q matches the 3rd of N.*
2. *For any remaining VeryOrigTable of N, it must match one of Q's VeryOrigTable. For each such match, we look at the matching column positions. As well, we get a mapping, such as* $(1, 2, 3, 4) \rightarrow (1, 2, 5, 3)$. *The mapping must be identical to that is recorded for the first VeryOrigTable.*

7.2.3 Choose Optimal Sharable Node

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

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

Chapter 8

Projection Management

As discussed in Chapter 5, to save materialization space and execution time, minimum column projection is performed. It projects only the necessary columns to a new node from its parents. The necessary columns include those in the final results and those needed for further evaluation. The process becomes intricate when sharing is considered. When a node is shared, it may not contain all the columns needed for the new query. Then extra columns will be added to the node and possibly to its ancestors by *projection enrichment*. Projection enrichment is necessary to exploit sharing opportunities with a new query that needs additional columns being added to the shared node.

To perform such operations, we need to index the projected column information and develop algorithms to search and update the index. This chapter describes the column indexing schema in Section 8.1 and the related algorithms in Sections 8.2 and 8.3.

8.1 Column Indexing

Projected columns are indexed in a set of system catalog relations, as shown in Figures 8.3, 8.5, 8.4, and 8.6. These relations describe the column transformations between nodes.

A projected column in a node N is either a simple projection of an individual column, or a projection of a complex expression, from one of N's direct parents. The projected column is also associated with N's other ancestors, as it is projected from the ancestors all the way down to N. Figure 8.1 shows the column transformations between the nodes of Example 1.1.

First, we look at direct transformations. For example, column *tranid*_20 in $J2$ is projected from *tranid*_14 in $J1$; *tranid*_14 in $J1$ is projected from *tranid*_9 in $S2$; *tranid*_9 in $S2$ is projected from *tranid*_1 in $S1$; and *tranid*_1 in $S1$ is projected from *tranid* in F. We also look at the indirect transformations. For example, both *tranid*_14 and *tranid*_15 are transformed from *tranid* in F.

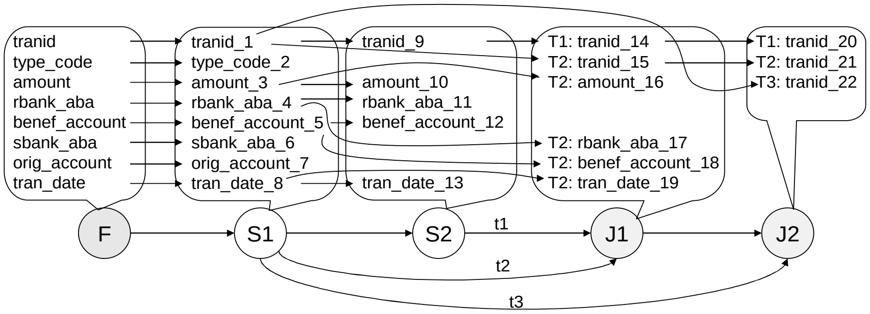


Figure 8.1: Projection for Example 1.1. The callouts show the projected columns projected for nodes S1, S2, J1, J2 and the mapping from their direct parent nodes. The identification number is appended to the column names to make universal column names.

For a **projected column** C in node N, e.g. *tranid*_14 in $J1$, a **base column** C_b is a column in N's ancestor N_a from which C is projected. We are interested in two base columns. The first is the base column C_d in N's direct parent N_d, called **direct**

base column. And the second is the base column C_v in N's veryorig ancestor N_v, called **veryorig base column**. If N_v is N's SVOA table, C_v is called **SVOA base column**. If N_v is N's JVOA table, C_v is called **JVOA base column**. If node N has both SVOA and JVOA tables, then C has both SVOA and JVOA base columns. Similar veryorig base column notions apply to aggregate node columns and set operator node columns.

Similar to topology indexing, we observed that for a projected column, we need to know both its direct base column and its veryorig base column(s). The direct base column is used for code generation, and the base columns are used for searching. Thus both mappings are recorded in the system catalog.

Consistent to topology indexing, the column mappings are also confined to join-level-locality, aggregate-locality, or set-operator-locality for consistent searching and management.

Since the pieces of information needed to index selection node columns, join node columns, aggregate node columns, and set operator node columns are all different to each other, we have different indexing schemas for them. The differences will be shown in following sub-sections.

A projected column in a node N is either carried along from an individual column in N's ancestor, called **simple projection**, or is a complex expression from N's ancestor(s), called **expression projection**. When the column is an expression projection, we index the canonicalized expression, so searching can be done easily. The expression is called **projection expression**.

8.1.1 Selection and Join Node Column Indexing

Both veryorig base columns and direct base columns should be recorded for projected columns in selection or join nodes. Veryorig base columns are used for searching available columns, since the columns are expressed in the form of veryorig base columns in the new query. And the direct base column is used for constructing code and tracing back during projection enrichment.

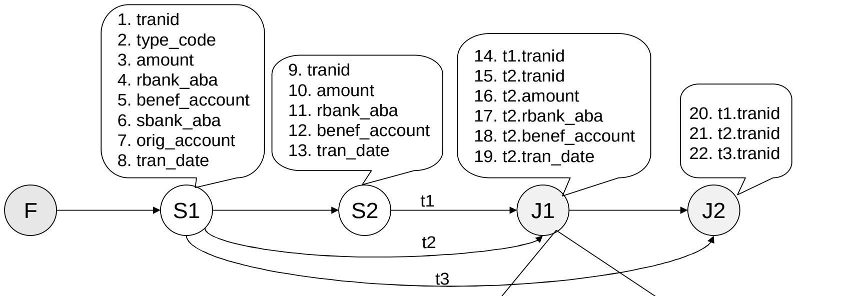
Due to different information structures, simple projection columns and expression projection columns are indexed in different column indexing relations, namely, JoinSimpleColumnNameMap, JoinExprColumnNameMap, SelSimpleColumnNameMap, and SelExprColumnNameMap. However, the SimpleColumnNameMaps should not only index the simple projection columns but also the columns that appear in the projection expressions.

Self-join nodes may have column name conflicts. For example, in Figure 8.2, $tranid\_14$ and $tranid\_15$ have the same JVOA base column form $tranid$ in F. They are distinguished by different table aliases in SQL. To materialize the join results, we need to assign two different projected column names ($tranid\_14$ and $tranid\_15$) to them. In ARGUS, we use unique **universal column names** and $TableAlias$ attribute to resolve such name conflict.

8.1.2 Aggregate Node Column Indexing

Aggregate node column indexing is simpler. Constrained by aggregate semantics, the projected column of an aggregate node must be either a group expression (a column or an expression of columns) or an aggregate function expression. A projected aggregate function is either a result function specified in the selection-clause or a basic aggregate function needed for incremental aggregation. If a projected column is a group expression, we also record its GroupExpressionID.

Since aggregates are unary relation operators, there is no name conflict. Therefore, we use the same universal column name for a projected expression along the vertical-expanded node chains, which simplifies the indexing and searching procedures. With this simplification, and with the fact that any aggregate node has a single direct parent without ambiguity, we do not need to record the direct parent information.



JoinSimpleColumnNameMap

Univ Column Name	Current Table name	Direct Orig Table Name	DirectOrig Column Name	Direct Orig Table Alias	Direct Orig Using Alieas	Very Orig Table Name	VeryOrig Column Name	Very Orig Table Alias	Very Orig Using Alias
tranid_14	J1	S2	tranid_9	Null	No	F	tranid	T1	Yes
tranid_15	J1	S1	tranid_1	Null	No	F	tranid	T2	Yes
amount_16	J1	S1	amount_3	Null	No	F	amount	T2	Yes
rbank_aba_17	J1	S1	rbank_aba_4	Null	No	F	abank_aba	T2	Yes
benef_account_18	J1	S1	benef_account_5	Null	No	F	benef_account	T2	Yes
tran_data_19	J1	S1	tran_date_8	Null	No	F	tran_date	T2	Yes
…	…	…	…	…	…	…	…	…	…

Figure 8.2: Projection for Example 1.1. The callouts show the columns projected for nodes S1, S2, J1, J2 with the preceding identification numbers. The identification number is appended to the column names to make universal column names. The JoinSimpleColumnNameMap shows the entries of the J1 columns.

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected join node name N. The projected column is in this node.
DirectOrigTableName	N's direct parent N_d.
DirectOrigColumnName	C's direct base column C_d. It is in N_d
DirectOrigTableAlias	TableAlias of N_d in the direct PredSet. Null if not self-join.
DirectOrigUsingAlias	A binary flag indicating whether the direct PredSet is a self-join.
VeryOrigTableName	N's join veryorig table N_v.
VeryOrigColumnName	C's join veryorig base column C_v.
VeryOrigTableAlias	TableAlias of N_v in the join veryorig PredSet. Null if not self-join.
VeryOrigUsingAlias	A binary flag indicating whether the join veryorig PredSet is a self-join.
OnlyInExpression	A binary flag indicating whether the projected column only appears in a selection expression.

Figure 8.3: JoinSimpleColumnNameMap: column indexing scheme for join node simple projections.

8.1.3 Set Operator Node Column Indexing

A set operator node is the result of a set operation on two or more parent nodes. The set operator node column indexing is thus similar to join node column indexing in terms of

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected join node name N. The projected column is in this node.
DirectExpressionCanonical	The canonical expression text in direct base columns.
DirectOrigTableName1	N's first direct parent N_{d1}.
DirectOrigTableName2	N's second direct parent N_{d2}.
DirectOrigUsingAlias	A binary flag indicating whether the direct PredSet is a self-join.
VeryOrigExpressionCanonical	The canonical expression text in join veryorig base columns.
VeryOrigTableName1	N's first join veryorig parent N_{v1}.
VeryOrigTableName2	N's second join veryorig parent N_{v2}.
VeryOrigUsingAlias	A binary flag indicating whether the join veryorig PredSet is a self-join.

Figure 8.4: JoinExprColumnNameMap: column indexing scheme for join node expression projections.

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected join node name N. The projected column is in this node.
DirectOrigTableName	N's direct parent N_d.
DirectOrigColumnName	C's direct base column C_d. It is in N_d.
SelectionVeryOrigTableName	N's selection veryorig table N_{vs}.
SelectionVeryOrigColumnName	C's selection veryorig base column. It is in N_{vs}.
JoinVeryOrigTableName	N's join veryorig table N_v.
JoinVeryOrigColumnName	C's join veryorig base column C_v.
JoinVeryOrigTableAlias	TableAlias of N_v in the join veryorig PredSet. Null if not self-join.
JoinVeryOrigUsingAlias	A binary flag indicating whether the join veryorig PredSet is a self-join.
OnlyInExpression	A binary flag indicating whether the projected column only appears in a selection expression.

Figure 8.5: SelSimpleColumnNameMap: column indexing scheme for selection node simple projections.

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected join node name N. The projected column is in this node.
DirectExpressionCanonical	The canonical expression text in direct base columns.
DirectOrigTableName	N's direct parent N_d.
SelectionVeryOrigCanonical	The canonical expression text in selection veryorig base columns.
SelectionVeryOrigTableName	N's selection veryorig table N_{vs}.
JoinVeryOrigExpressionCanonical	The canonical expression text in join veryorig base columns.
JoinVeryOrigTableName1	N's first join veryorig parent N_{v1}.
JoinVeryOrigTableName2	N's second join veryorig parent N_{v2}.
JoinVeryOrigUsingAlias	A binary flag indicating whether the join veryorig PredSet is a self-join.

Figure 8.6: SelExprColumnNameMap: column indexing scheme for selection node expression projections.

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected aggregate node name N. The projected column is in this node.
ExpressionCanonical	The canonical expression in veryorig base columns.
VeryOrigTable	N's veryorig table N_v.
GroupExpressionID	The GroupExpressionID if the column is a group expression. Null if not.
ColumnType	A binary flag indicating whether the column is a group expression or not.
SeqID	The position of the column in N.

Figure 8.7: GroupColumnNameMap: column indexing scheme for aggregate node columns.

the need to record both direct base columns and veryorig base columns. However, since set operator nodes do not have interleaving selection-type nodes, the column indexing is simpler, particularly without SVOA-like base column information to be indexed.

In a set operator node, the projected columns are just the set of columns specified in the selection-clause. The order in which the projected columns are specified in the selection clause must be preserved for the set operation. Thus the column positions are also recorded.

A set operation may operate on a parent node multiple times (similar to self-joins). And for some set operator, such as Difference (MINUS), the position of the parent node appearing in the query must be preserved. Therefore, we need a position ID for each parent node operand to uniquely distinguish it.

UnivColumnName	Universal Column Name C. The projected column name.
TableName	Projected set operator node name N. The projected column is in this node.
DirectOrigTableName	N's direct parent N_d.
DirectOrigUnionSubID	The set operation ID for N_d.
DirectOrigColumnName	C's direct base column C_d. It is in N_d.
VeryOrigTableName	N's veryorig table N_v.
VeryOrigUnionSubID	The set operation ID for N_v.
VeryOrigColumnName	C's direct base column C_v. It is in N_v.
UnivColumnSubID	The position of the column in N.

Figure 8.8: UnionColumnNameMap: column indexing scheme for set operator node columns.

8.2 Minimum column projection

Minimum column projection states that we should project only the minimal set of columns in a node, namely the necessary columns, to save space and execution time.

When a selection or join node is created to compute a where-clause PredSet, the necessary columns are all the columns in the selection-clause, the groupby-clause, the having-clause, and the columns in all the remaining predicates of the where-clause.

When an aggregate node is created, the necessary columns are the columns projected from the group expressions and the columns projected from the basic aggregate functions or the final aggregate functions required by the query.

When a set operator node is created, the necessary columns are just the ones specified by the selection-clause.

8.3 Projection enrichment

When query networks are not shared, the minimum column projection can be easily implemented for each type of nodes. However, when sharing is applied, much more intricacy arises. Given a node N that can be shared to evaluate a new query Q, N may not contain all the necessary columns needed by Q.

Assume[1] the set $C^u(N)$ of projected columns of N is projected from a subset of the columns in N's veryorig table(s) $\{N_v\}$. The subset is denoted as $Subset(C^u(N_v), N)$, which is the subset of $C^u(N_v)$.

Assume on node N, the necessary column set for Q in the form of N's veryorig base columns is $Columns(C^u(N_v), Q, N)$. It can be shown that this set of columns is always a subset of $C^u(N_v)$, thus $Columns(C^u(N_v), Q, N)$ can be presented as $Subset(C^u(N_v), Q, N)$. Since N is sharable to evaluate Q, N's veryorig tables $\{N_v\}$ are also sharable to evaluate

[1] $C^u(N)$ presents the set of columns in the form of its projected column name in N. Which N's veryorig table(s) refer to depends on the N node type and the sharing context.

Q, therefore $\{N_v\}$ must have projected all the necessary columns for Q. This theorem confines the projection enrichment within one join-level.

When $Subset(C^u(N_v), Q, N) \nsubseteq Subset(C^u(N_v), N)$, or N does not contain all the columns requested by Q, the projection enrichment has to be performed on N to make it trully sharable for Q. Since any nodes between N and N_v may or may not contain all the columns requested by Q, the chained projection enrichment has to be applied on each of them. When join is involved, the procedure may be branched into two ancestor chains.

Generally, the projection enrichment is a recursive branched chaining process to add necessary columns to nodes to allow an existing node to be shared to evaluate a new query. The process starts at the node N that is identified for sharing, and stops at or before N's veryorig tables N_v. Which type of veryorig tables to be used depends on the sharing type and the query context.

Given a node N to be shared for a query Q, and a set of columns $Subset(C^u(N_v), Q, N)$ that should appear in the node to obtain final results of Q, projection enrichment finds which columns are available in N, and which are not. The columns that are not in N, $Subset(C^u(N_v), Q, N) - Subset(C^u(N_v), N)$, should be added to N. These added columns may or may not appear in N's direct parent tables. If not, they have to be added into the direct parent tables as well. This chained projection enrichment has to be processed recursively until the direct parent contain all the requested columns.

One major challenge is to identify the direct parent of a column given its VeryOrigTable column form. This is clearly easy when there are no branches (selection or aggregate nodes). It becomes tricky when joins are involved, particularly when self-join and selection chains are present. We distinguish three cases.

1. When on both veryorig tables and direct parent tables, the joins are self-joins, shown in Figure 8.10(a), then the direct parent tables inherit their ancestors' STAs, and the direct parent of the column can be identified.

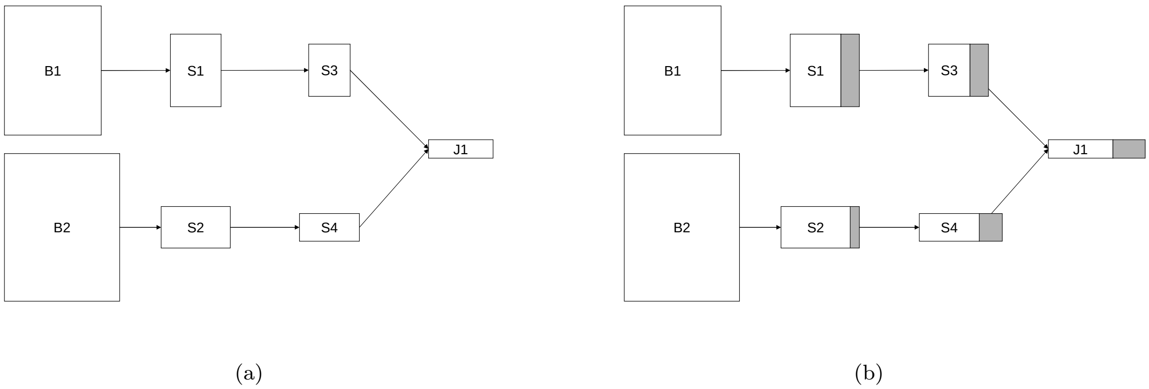
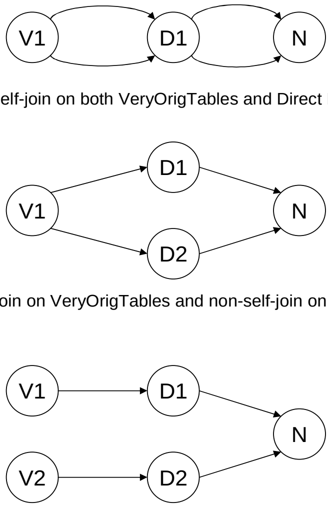


Figure 8.9: Chained Projection Enrichment. (a) The join node J1 is identified as sharable for a new query Q. (b) More columns need to be added into J1. Further they have to be added to J1's ancestors S3, S1, S4, and S2, as well.

2. When the join is a self-join on the veryorig tables, and is a non-self-join on the direct parent tables, show in Figure 8.10(b), we specify that the direct parents are indexed in the order of their STAs in JoinTopologyTable for the new join node N. Particularly the direct parent with STA $T1$ is indexed as $OrigTableName1$, and the other with STA $T2$ as $OrigTableName2$. Then the direct parent can be identified by matching the indexing position and the STA.

3. When the join is a non-self-join on both veryorig tables and direct parent tables, shown in Figure 8.10(c), the direct parent is the one that is a descendent of the column's veryorig table.



(a) Self-join on both VeryOrigTables and Direct Parents

(b) Self-join on VeryOrigTables and non-self-join on Direct Parents

(c) Non-self-join on both VeryOrigTables and Direct Parents

Figure 8.10: Three cases of joins from which the direct parent of a column needs to be identified given the column's veryorig table.

Chapter 9

Parsing, Plan Instantiation, and Code Assembly

This chapter describes the remaining operations necessary to construct shared query networks. They are query parsing described in Section 9.1, plan instantiation in Section 9.2, and code assembly in Section 9.3. Query parsing is the first step of the processing that transforms the new query in text form to a logical tree structure. Once the parse tree goes through the canonicalization and PredSet formation, it is fed to the IMQO module. Given a sharing plan generated by the IMQO, or a construction plan generated by the query optimizer, the plan instantiator performs necessary operations on the existing query network to carry out the intended computations specified in the plan, updates the system catalog to reflect the changes, and rewrite the query parse tree to reference the results of the computations. The code assembler sorts the executable code blocks generated by the plan instantiator and wraps them up into a set of stored procedures which present the executable code of the shared query network.

9.1 Query Parsing

The query parser parses a query Q into a parse tree. It takes the query Q as input, and outputs a parse tree. Figure 9.1 shows the structure and the data flow of the parser. It contains a SQL parsing module and a lexer module. The SQL parsing module is a Perl package generated by a compiler compiler Perl-byacc [36] based on a publicly-available SQL grammar [80]. The lexer module is also a Perl package [120]. The lexer is called by the SQL parsing module to tokenize the input query. The stream of tokens is fed to the SQL parsing module to generate the parse trees. The tools we used, namely, Perl-byacc, Perl Lexer, and the SQL grammar, were all from open sources with slight modifications.

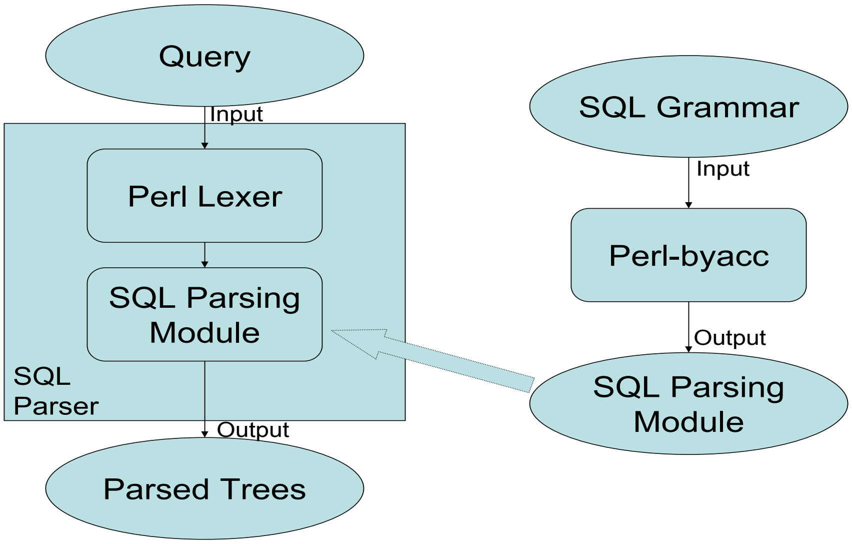


Figure 9.1: Building the SQL Parser

The where-clause parse tree is further processed by classifying predicates into PredSets. Figure 9.2 shows the schematic subtree of the query in Example 1.1. Predicates are classified based on the tables they use. For the subtree in Figure 9.2, the classifications are shown in Figure 9.3.

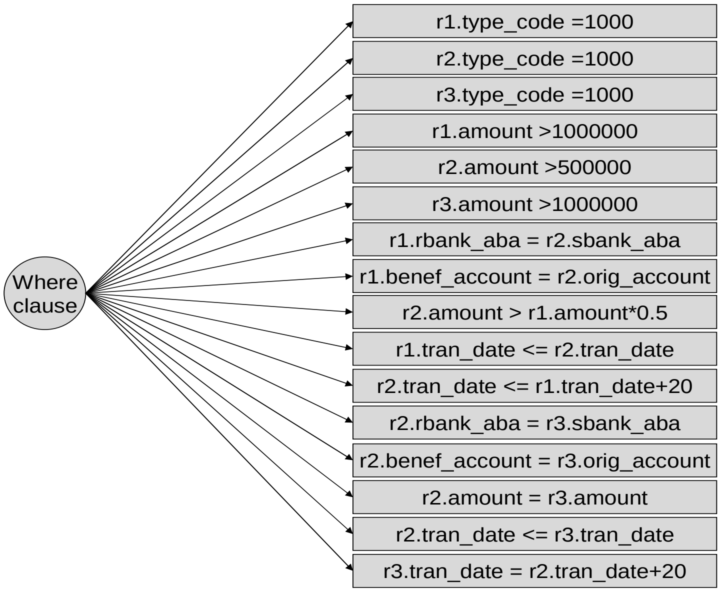

Where clause
r1.type_code =1000
r2.type_code =1000
r3.type_code =1000
r1.amount >1000000
r2.amount >500000
r3.amount >1000000
r1.rbank_aba = r2.sbank_aba
r1.benef_account = r2.orig_account
r2.amount > r1.amount*0.5
r1.tran_date <= r2.tran_date
r2.tran_date <= r1.tran_date+20
r2.rbank_aba = r3.sbank_aba
r2.benef_account = r3.orig_account
r2.amount = r3.amount
r2.tran_date <= r3.tran_date
r3.tran_date = r2.tran_date+20

Figure 9.2: The *where_clause* parse tree for Example 1.1

9.2 Plan Instantiation

A plan is either a sharing plan generated by the IMQO module or an optimized construction plan generated by the query optimizer. It describes how to expand the existing query network to obtain the intermediate and final results of Q. The process to follow the plan and expand the query network is called plan instantiation.

Shown in Figure 9.4, plan instantiation is comprised of plan traversal, node instantiation, and query rewriting. This chapter will describe these procedures.

The node instantiation is comprised of four sub-procedures, predicate/PredSet or group-expression indexing, topology indexing, column projection, and code generation. Predicate/PredSet indexing is described in Section 6.2. Group-expression indexing is described in Section 7.1. Topology indexing is described in Sections 6.2, 7.1, and 7.2. Column projection is described in Chapter 8. And code generation is described in Chapter 4. This section describes the control flow and the data flow of the node instantiation and the

$$
\begin{array}{ll}
r1: & \\
 & r1.\mathit{type\_code} = 1000 \\
 & r1.\mathit{amount} > 1000000 \\
 & \\
r2: & \\
 & r2.\mathit{type\_code} = 1000 \\
 & r2.\mathit{amount} > 500000 \\
 & \\
r3: & \\
 & r3.\mathit{type\_code} = 1000 \\
 & r3.\mathit{amount} > 500000 \\
 & \\
r1,\ r2: & \\
 & r1.\mathit{rbank\_aba} = r2.\mathit{sbank\_aba} \\
 & r1.\mathit{benef\_account} = r2.\mathit{orig\_account} \\
 & r2.\mathit{amount} > 0.5 * r1.\mathit{amount} \\
 & r1.\mathit{tran\_date} <= r2.\mathit{tran\_date} \\
 & r2.\mathit{tran\_date} <= r1.\mathit{tran\_date} + 20 \\
 & \\
r2,\ r3: & \\
 & r2.\mathit{rbank\_aba} = r3.\mathit{sbank\_aba} \\
 & r2.\mathit{benef\_account} = r3.\mathit{orig\_account} \\
 & r2.\mathit{amount} = r3.\mathit{amount} \\
 & r2.\mathit{tran\_date} <= r3.\mathit{tran\_date} \\
 & r3.\mathit{tran\_date} <= r2.\mathit{tran\_date} + 20
\end{array}
$$

Figure 9.3: Predicate classifications for Example 1.1. Classify them into PredSets.

related algorithms to compute the information flowing through these sub-modules.

9.2.1 Plan traversal

This subsection overviews the plan structure and the traversal process.

Both sharing plan and construction plan contain two pieces of information, topology of the new part, and the columns that should be projected for each node in the new part.

In the current implementation, the topology of a sharing plan is simple, since only one node N is provided for sharing, even though there may be one or more intermediate nodes between N and its original tables that the query references.

Construction plans are more complex. It is a tree. Each node presents the sub-plan of how to create a query network node. The tree shape presents the topology of the query

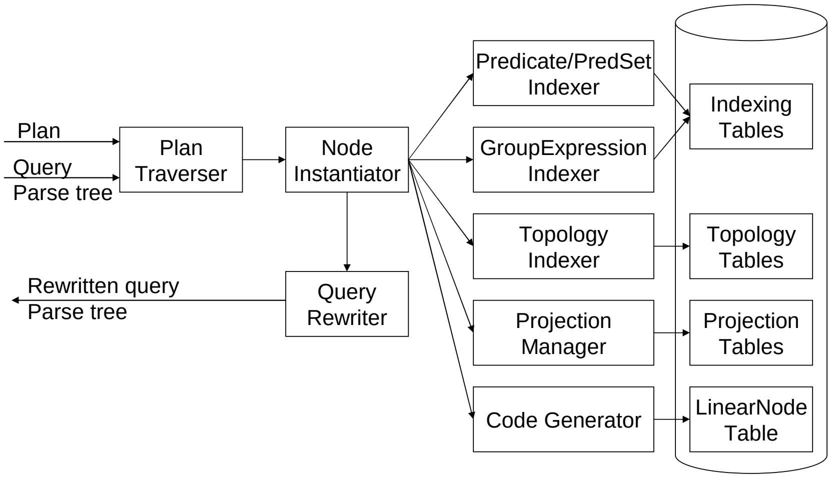


Figure 9.4: Plan Instantiation. Plan Instantiator traverses the plan, sends individual node create/update instructions to node instantiator, and calls query rewriter to rewrite the logical parse tree. Node instantiator calls sub-modules to index predicate/PredSets, group expressions, topologies, and columns, and to generate and store the code blocks for the node.

network part to be expanded. The root node presents the entire plan to obtain the final results for the new query Q.

We need to clarify the child-parent definition in the plan tree, since this is precisely opposite from the definition in query network. Query network is a DAG. Edges are pointed from parents to children. In a query network, we apply operators on parent nodes to produce the results of the child node. Thus the direction of an edge is from an operand node to a result node. A child may have multiple parents.

However, in a plan tree, the final result node is the root node, which is conceived as the ancestor of the remaining nodes according to a widely-accepted child-parent definition on trees. Figure 9.5 shows this difference. Figure 9.5(a) is the existing query network R, and (b) shows a tree-shaped construction plan that can be used to expand R, and Figure 9.6

shows the expanded query network where edge directions from the plan are reversed.

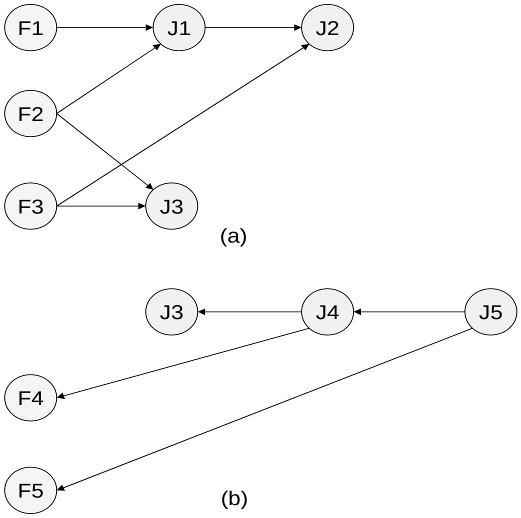


Figure 9.5: Child-parent in query networks and in plan trees. (a) The existing query network. Children are result tables. (b) The plan tree. Children are operand tables.

The plan traversal is a recursive bottom-up tree traversal procedure. A node (sub-plan) is traversed (processed or instantiated) after all of its children nodes (sub-plans) are traversed (processed or instantiated). In another word, a sub-plan is only instantiated after all of its children sub-plans are instantiated.

Instantiation of a sub-plan involves creating a new node or updating an existing node. This process is called node instantiation. After the node instantiation, the query is rewritten to reference the node table.

9.2.2 Node Instantiation

Node instantiation computes and records all the information related to the new or updated node. Different types of nodes require different instantiation procedures. However, all the procedures involve topology indexing, column indexing, and code generation. Besides these

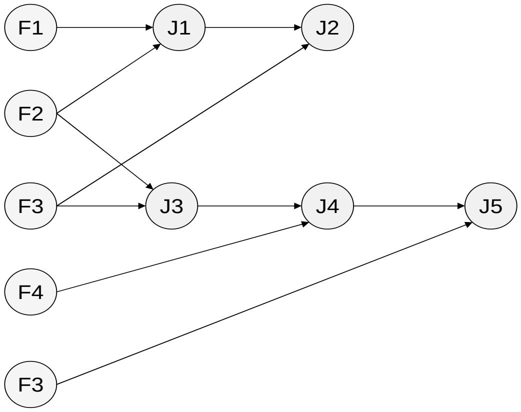


Figure 9.6: Child-parent in query networks and in plan trees. When the plan tree is instantiated to the query network, the edge directions are reversed.

common indexing sub-procedures, a selection or join node also involves predicate/PredSet indexing (see Section 6.2); an aggregate node also involves group expression indexing (see Section 7.1); and a set operator node does not involve any specific indexing.

When creating a new or updating an existing selection/join node, the SVOA and JVOA PredSets and the direct PredSet are computed by the PredSet conversion, union, and difference operations. These PredSets are indexed. Then the topology entry of the node is created or updated. And finally, the column information is indexed in the column system catalog relations. The column indexing also contains a chained projection enrichment procedure to add necessary columns to the ancestors of the node. Given the indexed information, the code generator is called to generate the code blocks for the node, and the code blocks are stored in the LinearNodeTable.

When creating a new or updating an existing aggregate node, the group expressions, and the group expression sets (dimension sets) are indexed. Then the topology entry of the node is created or updated. And the columns (group expressions and aggregate functions) are indexed. Given the indexed information, the code generator is called to generate the code blocks for the node, and the code blocks are stored in the LinearNodeTable.

When creating or updating a set operator node, the topology and column information are indexed. Given the indexed information, the code generator is called to generate the code blocks for the node, and the code blocks are stored in the LinearNodeTable.

9.2.3 Query rewriting

Assume the new query Q is evaluated from a set of tables $\{M_v\}$. When a new node N is created or when an existing sharable node N is selected to provide the intermediate results for Q, Q's parse tree should be rewritten to reference N to reflect such progress.

Assume N's veryorig table set is $\{N_v\}$, where $\{N_v\}$ is a subset of $\{M_v\}$. The rewriting procedure replaces the references to $\{N_v\}$ and their columns by the references to N and its columns. It first rewrite Q's from-clause by replacing the reference to $\{N_v\}$ with N. Namely, $\{M_v\} \rightarrow (\{M_v\} - \{N_v\}) \cup N$. When there are self-joins or set operations on the same tables, tables within $\{N_v\}$ and $\{M_v\}$ are distinguished by their unique table aliases. Then it replaces the related column references. For any column that is in one of $\{N_v\}$, replace it with its new projected column name in N. The mapping between the old column names to the new column names is constructed when N is created.

9.3 Code Assembly

The query network is evaluated in a linear fashion, and the nodes need to be sorted. The only sorting constraint is that the descendant nodes must follow their ancestor nodes, 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[1], 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 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 9.1 *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

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

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 contradicts 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 node 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 contradicts to the fact proved in the first step.

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

With JoinLevel and SequenceID defined, a 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.

Chapter 10

Evaluation

We conduct experiments to understand the effectiveness of various techniques implemented in ARGUS. Particularly, we evaluate the effectiveness of incremental evaluation methods and optimization techniques on selection-join-projection queries, the effectiveness of incremental aggregation and IMQO on aggregate queries, and the effectiveness of IMQO techniques on SJP queries. The evaluated IMQO techniques include canonicalization, join sharing, and the two sharing strategies, match-plan and sharing-selection [1].

The results show that every individual technique lead to significant performance improvement either in general or at least for some specific types of queries. As a whole, the system provides acceptable performance for continuously matching large-scale queries. The analysis of the results also raises new questions and points to new research directions.

In this chapter, we first discuss the overall experiment setting including data, query, and the simulation setting, then we present the experiments and results.

[1] We had difficulty to directly compare ARGUS with other publicly available stream systems, such as STREAM and NiagaraCQ. For example, STREAM does not support common computation identification among multiple queries, and NiagaraCQ uses XML data schema and XQL query language.

10.1 Experiment Setting

The experiment setting is comprised of three components, data, query, and the simulation setting. In this section, we describe the data sources and their generation, the query generation and the query characteristics, and how the data and queries are used in the experiments.

10.1.1 Data

We use two databases, the synthesized FedWire money transfer transaction database (FED), and the anonymized Massachusetts hospital patient admission and discharge record database (MED). Both databases have a single stream with timestamp attributes. Appendix A describes the schemas.

FED is a synthesized database containing 500006 FedWire money transfer transactions. The schema contains all the real transaction data attributes. The first 500000 records are generated according to the real transaction statistics. The last 6 records are generated to bring some linkages between transactions and to satisfy some experiment queries which simulates the very rare alert-triggering conditions.

MED is a real medical database of 835890 inpatient admission and discharge records. Personal information, such as name and address, are dropped for privacy protection, but demographic information, such as age, gender, race, and residency township are retained.

10.1.2 Queries

We created a query repository for FED and MED databases. It contains several query sets for different experiments. These queries were generated systematically in three steps.

Firstly, interesting queries arising from applications are formulated manually as query seeds or query categories. The seeds cover a wide range of query types, including selections, joins, aggregates, set operators, and their combinations. The seed queries vary in several ways and present some overlap computations. For example, there are SJP queries and aggregate queries, and some aggregate queries aggregate on the results of SJP queries and can share from them. Appendix B presents the typical seed queries used in our experiments. In the experiments, we also used variants of seed queries, e.g. queries with different number of joins. We describe these variants in Section 10.1.3.

Secondly, changeable query parameters, or the constants, are identified. The variants of seed queries, such as those with different number of joins but with similar join and selection predicates, are also considered as different values of changeable parameters. For example, for the query category as shown in Section 1.4, we used the variants of the query that differ in three dimensions. The first (number of joins) varies the self-joins from 2-way, up to 5-ways. The second (tracking direction) varies the placement of the money split join which indicates the binary tracking direction (look forward or backward) of the money flow. And the third defines two types of groups (one based on account numbers and the other on bank numbers) that the joined records are confined to. For example, two records join only if the first one transfers money from a certain type of bank/account and the second transfers to another type of bank/account. Further, we identify three more simple changeable parameters, the transfer amount, the join time window, and the amount split ratios between joined records. These parameters can be changed easily by value substitution.

Thirdly, more queries are generated by varying the parameters of the seed queries. The queries generated from the same query seed present overlap computations, such as subsumptions, and can be shared. The varying parameters are either selected manually or generated by random draws from a distribution of parameters.

10.1.3 Query Sets

We generated several query sets for different experiments on the FED and MED databases, as shown in Table 10.1. QIEFeds are used to evaluate incremental evaluation methods and query optimization techniques. QAggreFed and QAggreMed are used to evaluate incremental aggregation and IMQO on aggregate queries. And QIMQOSJPFeds and QIMQOSJPMed are used to evaluate IMQO on SJP queries.

Query Sets	Data Base	Purpose	Parameter Value Selection	Number of Variants	Number of Queries
QIEFed-Manual	FED	Incremental evaluation and query optimization	Manual	7	7
QIEFed-Uniform	FED	Incremental evaluation on SJP queries and query optimization	Random Draws	8	160
QAggreFed	FED	Incremental aggregation and IMQO on aggregate queries	Manual	21	350
QAggreMed	MED	Incremental aggregation and IMQO on aggregate queries	Manual	3	450
QIMQOSJPFed-Manual	FED	IMQO on SJP queries	Manual	16	768
QIMQOSJPFed-Uniform	FED	IMQO on SJP queries	Random Draws	16	1000
QIMQOSJPFed-Normal	FED	IMQO on SJP queries	Random Draws	16	1000
QIMQOSJPFed-GaussianMixture	FED	IMQO on SJP queries	Random Draws	16	1000
QIMQOSJPMed	MED	IMQO on SJP queries	Manual	8	565

Table 10.1: Query sets for evaluation.

QIEFed-Manual contains 7 queries, presented as Examples B.1-B.7 in Appendix B, and is used to evaluate incremental evaluation and query optimization techniques. This is the first query set developed, and its variants are used to develop other FED query sets.

QIEFed-Uniform contains 8 query variants and 160 queries, 20 for each variant. The first 4 variants are the 2-way up to 5-way joins derived from Examples B.1 and B.3. And the other 4 variants are Examples B.2, B.4, B.6, B.7. We did not include Example B.5 in this query set; the aggregate query is left to QAggreFed.

QIEFed-Uniform and three of QIMQOSJPFeds were generated by randomly selecting parameter values. Particularly, for each query set, the queries are drawn independently from a joint parameter distribution. We define that the joint distribution is on independent random variables, so the draws of the values of the parameters within one query are also independent. Therefore, the joint distribution can be presented as a set of marginal distributions on each parameter.

The changeable parameters on QIEFed-Uniform queries include *transfer amount, join time window, transaction date, and bank name.* The marginal distribution on a parameter P is the uniform distribution on a range on P. The ranges of *transfer amount* and *join time window* are described in Table 10.2. The *transaction date* range is the month that covers the dates in FED data, and the *bank name* range is a randomly-picked set of 10 banks.

Both QAggreFed and QAggreMed are used to evaluate incremental aggregation and IMQO on aggregate queries. They are generated by manually selecting parameter values.

QAggreFed has 3 query categories, 21 query variants, and 350 aggregate queries on the FED database. They monitor alerting accumulative amounts of certain types of transactions. The first category performs selection and join before the aggregation, and have 16 variants in three dimensions (4 numbers-of-joins * 2 tracking-directions * 2 group-by-options). The second category performs join after the aggregation, and have 4 query variants depending on whether grouping by the receiving or sending party and whether the party entity is the account or the bank. And the third category performs set operations after the aggregation.

QAggreMed has 3 query categories and 414 distinct aggregate queries on the MED database. We pad the first 36 queries from the set to the end of it to make QAggreMed contain 450 queries. These queries monitor multiple occurrences of various contagious diseases in or beyond local areas within or not within a given time window. They perform selections and joins, then aggregations, and finally set operations.

Both QIMQOSJPFeds and QIMQOSJPMed are used to evaluate the IMQO approach on SJP queries. There are four QIMQOSJPFed query sets and one QIMQOSJPMed query set.

Each of the four QIMQOSJPFed query sets is generated from the 16 variants of one query category. Two query examples from the category are shown in Section 1.4. The variants vary in three dimensions, 4 numbers-of-join * 2 tracking-directions * 2 group-confinements. Among the 4 QIMQOSJPFeds, QIMQOSJPFed-Manual is generated by manually selecting the parameters, and each of the remaining three, QIMQOSJPFed-Uniform, QIMQOSJPFed-Normal, or QIMQOSJPFed-GaussianMix, is generated by random draws from a distribution of the parameters.

QIMQOSJPFed-Manual contains 768 queries. They are generated from the combinatorial combinations of the parameter values shown in Table 10.2.

Parameters	Number of Joins	Tracking Direction	Group Confinement	Transfer Amount	Join Time Window	Amount Split Ratio
Values	2,3,4,5	forward, backward	account, bank	50000, 100000	5,10,20, 30,40,60	1,2,5,10
Ranges	[2, 5]	{forward, backward}	{account, bank}	[50000, 100000]	[5, 60]	[1, 10]

Table 10.2: QIMQOSJPFed Parameter Values and Ranges. Number of Joins: the number of 2-way joins. Tracking Direction: whether the query tracks the money flow forward or backward based on a pivot transaction. The pivot transaction is defined by selection predicates on the transaction type (1000), the Group Confinement, and the Transfer Amount, and is the earliest one of the joined records in the forward case or the latest one in the backward case. Group Confinement: the transactions to be joined use certain types of bank or certain types of account. Transfer Amount: the transfer amount that the pivot transaction must be above. Join Time Window: the time window in days that any two directly joined records must fall in. Amount Split Ratio: the number of splits that the pivot transaction can be split for forward or backward money flow.

Each of the three randomly-generated FED query sets contains 1000 queries. For QIMQOSJPFed-Uniform, the marginal distribution on a parameter P is the uniform distribution on the range $[min, max]$ defined in Table 10.2. For QIMQOSJPFed-Normal, the marginal distribution on P is the normal distribution with $\mu = (min + max)/2$ and $\sigma = 0.1 * (max - min)$. And for QIMQOSJPFed-GaussianMix, the marginal distribution on P is the two-Gaussian mixture model with the equal model probability, $\mu_1 = min + (max - min) * 0.2$, $\mu_2 = min + (max - min) * 0.8$, and $\sigma = 0.1 * (max - min)$. All drawn values are rounded to integers. And for the binary non-numerical parameters, the equivalent Bernoulli drawing is used.

QIMQOSJPMed has 8 query categories and 565 queries generated by manually selecting parameters. The majority is the 3-category 414 queries used in QAggreMed. The remaining 5 categories monitor the patient hospital transferring, different types of disease development over time, or the links between diseases and operations. Since the accurate personal information is not available, the patient matches among multiple records use only the demographic information and are not accurate, but nevertheless are useful to demonstrate large-scale query settings.

10.1.4 Experiment Setting

The experiments were conducted on two system configurations. At first, QIEFed-Manual was evaluated on an HP PC computer with single core Pentium(R) 4 CPU 1.7GHz and 512M RAM, running Windows XP and Oracle 9.2. Later, the system was updated, and all the remaining experiments were conducted on a DELL PC computer with single core Pentium(R) 4 CPU 3.00GHz and 1G RAM, running Windows XP and Oracle 10.1.

To simulate the streams, in the order of time, we take the first part (300000 records from FED and 600000 from MED) of the data as historical data, and simulate the arrivals of new data incrementally. Query networks are evaluated on 10 incremental data sets 11 times for each set. Each incremental data set contains 4000 new records. Most of the experiments were conducted on such simulation settings unless otherwise specified.

Given the historical data and one incremental data set, the system performance is measured in the total time of producing new results for all the currently active queries. The queries from different categories are added to the system in an interleaving fashion, and the performance is recorded at each hundred number of queries. Besides the figures shown in this chapter, most experiment results are also shown in tables in Appendix C.

10.2 Incremental Evaluation on SJP Queries and Query Optimization

The experiments presented in this section were conducted on two query sets, QIEFed-Manual and QIEFed-Uniform. QIEFed-Manual was evaluated on the earlier system configuration: HP PC Pentium4 1.7G CPU and 512M RAM, running Windows XP and Oracle 9.2; and QIEFed-Uniform was on the later updated configuration: DELL PC Pentium(R)4 CPU 3.00GHz and 1G RAM, running Windows XP and Oracle 10.1.

The QIEFed-Uniform evaluation is more concrete than QIEFed-Manual. The QIEFed-Uniform results are averaged over a set of randomly-generated queries, instead of on single queries. The reason that we still show the old evaluation on QIEFed-Manual is because the two DBMS versions on different hardware configurations behave differently which merits some discussion.

On QIEFed-Manual, we use the following two data conditions:

- Data1. Historical data: the first $300,000$ records. New data: the next $20,006$ records. This data set provides alerts for the queries being tested. Particularly, the 6 additional records in Data1 give at least one result for each of the query in QIEFed-Manual.
- Data2. Historical data: the first $300,000$ records. New data: the next $20,000$ records. This data set does not generate alerts for most of the queries being tested.

On QIEFed-Uniform, the historical data part is the same, and the new data part is the next 1000 records. To evaluate the effect of batch-size on incremental evaluation, we vary the batch-size of the new data part from 1000 records to 20000 records, as shown in Figure 10.2.

The query network for each individual query is generated and evaluated. It takes some time for query networks to initialize intermediate results, yet it is an one-time operation. Query networks provide incremental new results. On the other hand, when running the original SQL queries directly on the DBMS, we combine the historical (old) data and the new data (stream), and the performance measurement is the time to process the whole data sets.

10.2.1 Incremental Evaluation

This section compares the execution time of the incremental evaluation (Rete) and running the queries on DBMS directly (DBMS). The experiments were conducted on two query sets, QIEFed-Manual and QIEFed-Uniform.

In incremental evaluation, the query network for each individual query is generated with following configuration: no hidden condition is added to the original queries, and transitivity inference is turned on.

Figure 10.1 summarizes the results of running the queries Q1-Q7 on the two data conditions. For most of the queries, query networks with transitivity inference gain significant improvements over directly running the SQL queries on the DBMS.

Q4 and Q5 are the two queries involving aggregations. The Q4 query network has a join that computes on the large original table. This significantly slows down the incremental matching. And Q5 is an aggregate-then-join query and the query network did not apply incremental aggregation, thus there is no performance improvement. We show later that incremental aggregation will provide significant improvements to these types of queries.

Figure 10.2 shows the average execution times over 160 QIEFed-Uniform queries with different batch-sizes. This figure still shows that incremental evaluation is better than

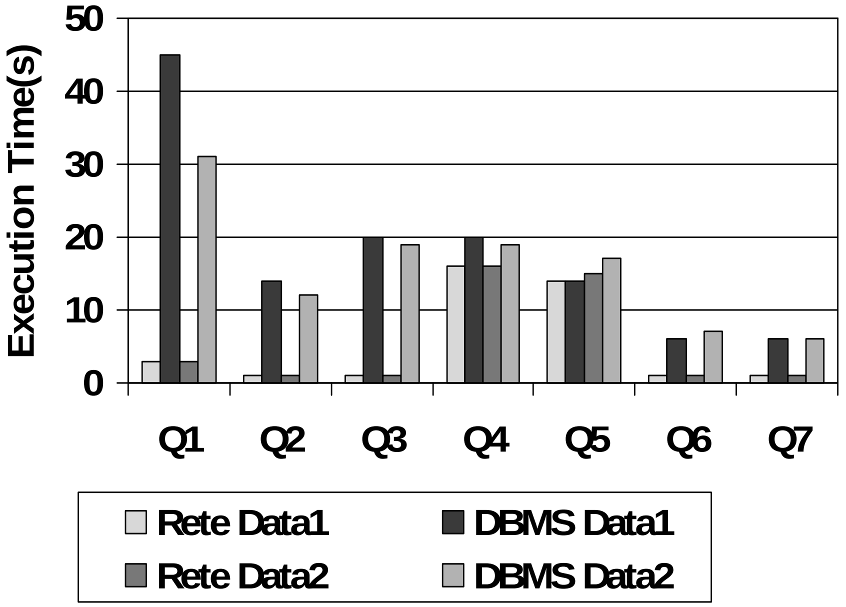


Figure 10.1: Execution times of QIEFed-Manual. This shows that incremental evaluation (Rete) is much faster than the naive approach (DBMS) for the majority of queries (Q1, Q2, Q3, Q6, and Q7) on both data conditions. Data1: the historical data is the first 300, 000 records, and the new data is the next 20, 006 records. Data1 provides alerts for the queries being tested. Data2: the historical data is the first 300, 000 records, and the new data is the next 20, 000 records. Data2 does not generate alerts for most of the queries being tested.

running queries directly on DBMS, but less significant. Two facts contribute to this insignificance. First, transitivity inference is automatically applied in the new DBMS version but not in the old one, as we will show later in Section 10.2.2. Second, the updated hardware improves the computation efficiency significantly over large-scale data as required by the DBMS approach, but not on the processing overhead as required by the incremental evaluation approach.

Figure 10.2 also shows the performance changes of the incremental evaluation over the changes of the batch size: the number of tuples in the new data part. Theoretically and ideally, the execution time should be linear to the batch-size. However, the figure shows two different behaviors at the two batch-size ends. At the small batch-size end (1000 to

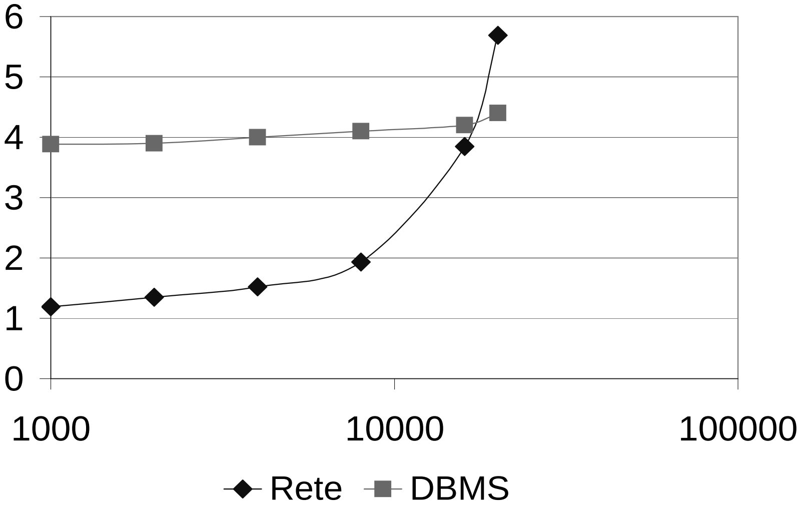


Figure 10.2: Execution times of QIEFed-Uniform. X-axis is the batch-size: the number of tuples in the new data part. Y-axis is the average execution time over the 160 queries. This shows that incremental evaluation (Rete) is much faster than the naive approach (DBMS) when the batch-size is small. But the execution time grows as the batch-size increases, and may exceed the DBMS approach.

8000), the execution time is sub-linear to the batch-size, because the majority of time is the overhead of the query network processing including loading the tables into memory. At the large batch-size end (8000 to 20000), the execution time becomes linear and finally higher than linear, because page swapping is involved. In practice, we want to tune the batch-size to provide fast response with computation efficiency. We will discuss this future work in Section 11.1.3.

10.2.2 Transitivity Inference

This section presents the results on transitivity inference. We show the experiments conducted on the two system configurations.

On the old system configuration with QIEFed-Manual, Q1 and Q3 are queries that benefit from transitivity inference. Figure 10.3 shows the execution times for these two examples. The inferred condition *amount* > 500000 is very selective with selectivity factor of 0.1%. Clearly, when transitivity inference is applicable and the inferred conditions are selective, a query network runs much faster than its non-TI counterpart and the DBMS approach.

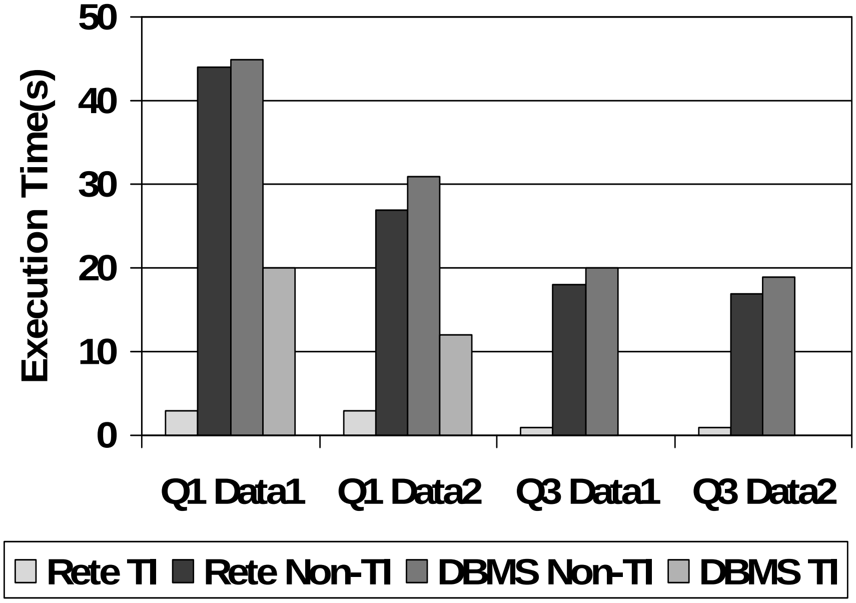


Figure 10.3: Effect of transitivity inference on QIEFed-Manual. This shows that transitivity inference leads to significant improvements to both Rete and DBMS. "Rete TI": Rete generated with transitivity inference. It achieves 20-fold improvement comparing to Rete Non-TI and DBMS Non-TI. "Rete Non-TI": Rete without transitivity inference. "DBMS Non-TI": original SQL query on DBMS. "DBMS TI": original SQL query with hidden conditions manually added on DBMS.

Note that in Figure 10.3, *DBMS TI* for Q1 is the time of running the SQL query with the inferred conditions manually added in. It runs significantly faster than the DBMS. This suggests that this type of complex transitivity inference is not applied in the DBMS query optimization, and can be implemented in a traditional DBMS to improve performance on evaluating traditional SQL queries.

On the new system configuration, we evaluated the transitivity inference on the first

4 query variants of QIEFed-Uniform, Q1 - Q4, (Section 10.1.3), as shown in Figure 10.4. Each variant has 20 queries. The historical data is the first 300000 records, and the new data part is the next 1000 records. The figure shows that transitivity inference leads to significant improvements to the incremental evaluation, but not to DBMS. This indicates that the new DBMS version implements the transitivity inference.

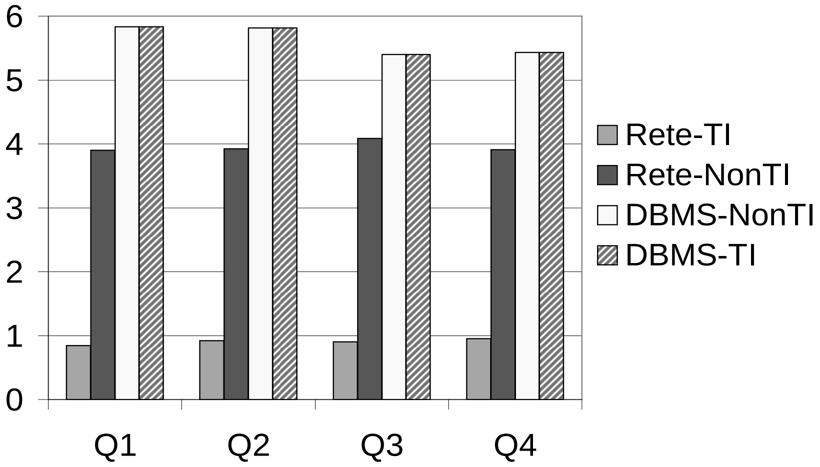


Figure 10.4: Effect of transitivity inference on QIEFed-Uniform's 4 query variants, Q1 - Q4, each of 20 queries. The historical data is the first 300000 records, and the new data part is the next 1000 records. The figure shows that transitivity inference leads to significant improvements to Rete, but not to DBMS. "Rete TI": Rete generated with transitivity inference. "Rete Non-TI": Rete without transitivity inference. "DBMS Non-TI": original SQL query on DBMS. "DBMS TI": original SQL query with hidden conditions manually added on DBMS.

10.2.3 Conditional Materialization

Conditional materialization is also tested on the two system configurations. The experiments were conducted by turning off the transitivity inference option. Then the queries have to join with large historical data parts.

We compared three configurations, Conditional, Non-Conditional, and DBMS. DBMS is running the query directly on the DBMS. Non-Conditional is the query network that materializes all selection and 2-way join predicate sets. Conditional is the query network that materializes all above mentioned predicate sets except those selection predicate sets with selectivity above 0.3.

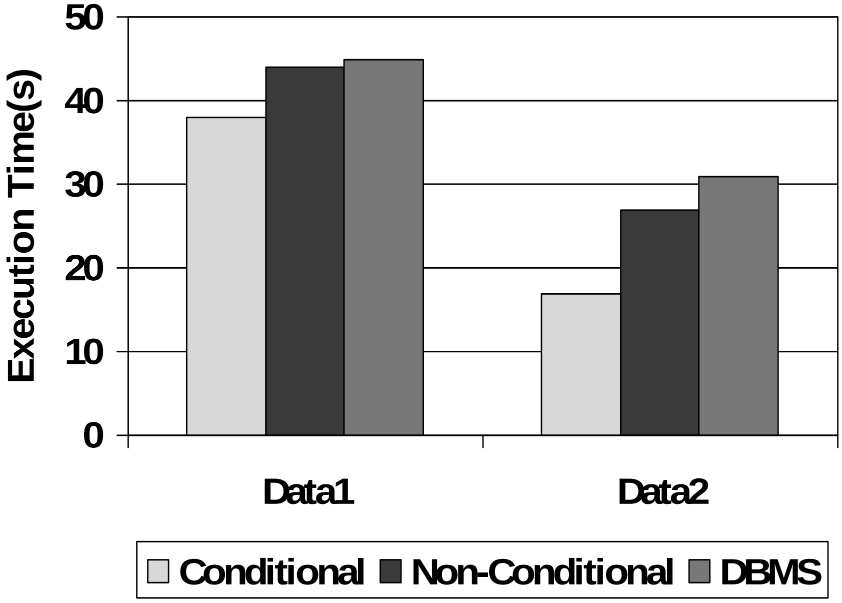


Figure 10.5: Effect of conditional materialization on QIEFed-Manual with transitivity inference turned off. Comparing the execution times of conditional materialization, non-conditional materialization, and running the original SQL on the DBMS.

On the old system configuration, in Figure 10.5, *Non-Conditional* presents the query network that still materializes the results of the non-selective selection predicates, and *Conditional* presents the query network that skip such inefficient materialization. The figure also shows the time of running the original SQL query on the DBMS. It is clear that if non-selective conditions are present, conditional materialization should be applied.

On the new system configuration, we evaluated the conditional materialization on the first 4 query variants of QIEFed-Uniform, as shown in Figure 10.6. The historical data

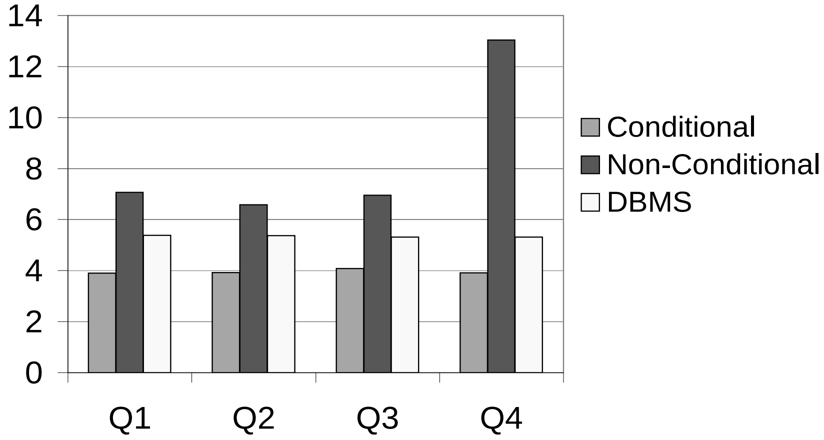


Figure 10.6: Effect of conditional materialization on QIEFed-Uniform's 4 query variants, Q1 - Q4, each of 20 queries with transitivity inference turned off. Historical data is the first 300000 records, and the new data part is the next 1000 records. In the Non-Conditional case, the materialization overhead is large enough to make the performance even worse than the DBMS approach. This problem is fixed by the conditional materialization.

is the first 300000 records, and the new data part is the next 1000 records. In the Non-Conditional case, the materialization overhead is large enough to make the performance even worse than the DBMS approach. This problem is fixed by the conditional materialization.

10.3 Incremental Aggregation and Aggregation IMQO

In this section, we present experiments to show: 1. the effect of incremental aggregation (*IA*) vs. the naive reaggregation approach (*NIA*), 2. the effect of IMQO on aggregate queries (*shared incremental aggregation, SIA*) vs. unshared aggregate queries (*non-shared incremental aggregation, NS-IA*), and 3. the performance changes of vertical expansion

(VE) and non-vertical expansion (NVE) with regard to $|S_N|$. $|S_N|$ is the incremental data size.

Table 10.3 shows the historical and new data part sizes on the FED and MED databases used in the experiments.

	FED	MED
$\|S_H\|$	300000	600000
$\|S_N\|$	4000	4000

Table 10.3: Evaluation Data for incremental aggregation.

We use 350 aggregate queries on FED and 450 aggregate queries on MED as discussed in Section 10.1.2. Some of these queries aggregate on selection and self-join results. Examples B.4, B.5, and B.8 to B.11 present some of these queries.

10.3.1 Incremental Aggregation

Figures 10.7 and 10.8 show the execution times of *IA* and *NIA* on each single query, and the ratio between them, *NIA/IA*. A *NIA/IA* ratio above 1 indicates better *IA* performance. Since there are more fluctuations on diversified MED queries, we show running averages over 20 consecutive queries in Figure 10.8 to make the plot clear. The queries are sorted in the increasing order of $AggreSize = |S_H(A)| * |A_H|$, shown on the second Y axis. $|S_H(A)|$ is the actual aggregation data size of the historical part after possible selections and joins. We tried to sort the queries based on two other metrics, $|S_H(A)|$, and $|S_H(A)| + |A_H|$. But they show less consistent trends to the time growth and the *NIA/IA* ratio. This indicates that the DBMS aggregation operator on S_H has time complexity of $|S_H(A)| * |A_H|$.

AggreSize indicates the query characteristics. There are about 250 queries in both FED and MED whose $|S_H(A)|$ is 0, shown to the left of the vertical cut lines and are sorted by the *NIA/IA* ratio. Unsurprisingly, their execution times are very small. The small time fluctuations are caused by the DBMS file caching.

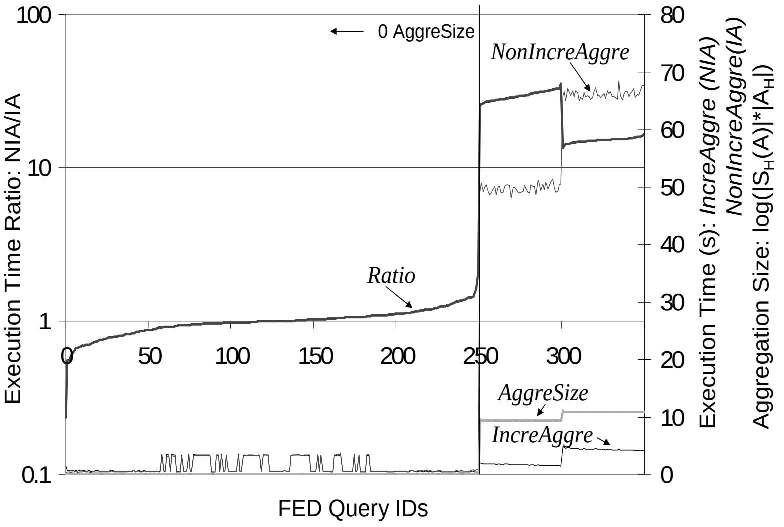


Figure 10.7: Single FED compares the execution times of incremental aggregation (IncreAggre, IA) and non-incremental aggregation (NonIncreAggre, NIA) on individual Fed aggregate queries. The x-axis is the query IDs sorted by the $AggreSize = |S_H(A)| * |A_H|$, which estimates the IA cost. The figure also shows the performance Ratio between the IA and the NIA: (IA execution time)/(NIA execution time).

For the remaining queries, incremental aggregation is better than non-incremental aggregation. Particularly, it gains more significant improvements when the aggregation size is large. These large-size queries dominate the execution time in multiple-query systems. Thus significant improvements on such queries are significant to the whole system performance, as shown in Table 10.4.

	FED 350Q	MED 450Q
IA	662	316
NIA	6236	938

Table 10.4: Total execution time in seconds for incremental aggregation (IA) and non-incremental aggregation (NIA).

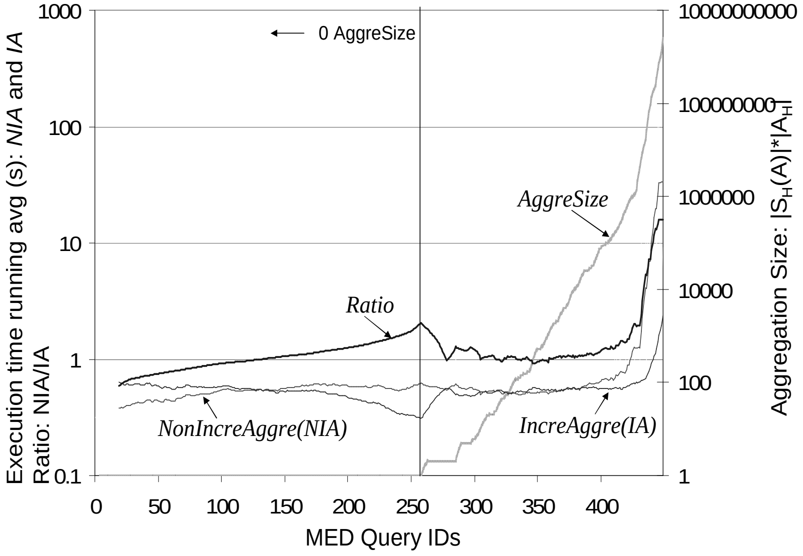


Figure 10.8: Single MED compares the execution times of incremental aggregation (IncreAggre, IA) and non-incremental aggregation (NonIncreAggre, NIA) on individual Fed aggregate queries. The x-axis is the query IDs sorted by the $AggreSize = |S_H(A)| * |A_H|$, which estimates the IA cost. The figure also shows the performance Ratio between the IA and the NIA: (IA execution time)/(NIA execution time).

10.3.2 IMQO on Aggregate Queries

Figures 10.9 and 10.10 show the total execution times of *SIA* and *NS-IA* by scaling over the number of queries. Clearly, incremental sharing provides improvements, particularly on FED where queries share more overlap computations.

It is interesting to note that the benefit of IMQO is much more significant on FED than on MED. Similar gap is observed on pure SJP queries; see Section 10.4. This is because the system does not recognize certain sharable computations on MED queries. Particularly, the majority MED queries involve certain type of disease (selection predicates on the disease category), but have same join predicates. The selection predicates from different queries specify a set of disjoint disease category ranges. Ideally, the system can create a selection node which selects all the tuples that are in the minimum cover of the disjoint ranges (the minimum consecutive range that covers each of the ranges), perform just one join from the selection node, and further select the tuples in each disease range from the join result. Currently, the system does not find the minimum range cover. We should note that the range cover can only be identified on ordered data types, such as numerical values.

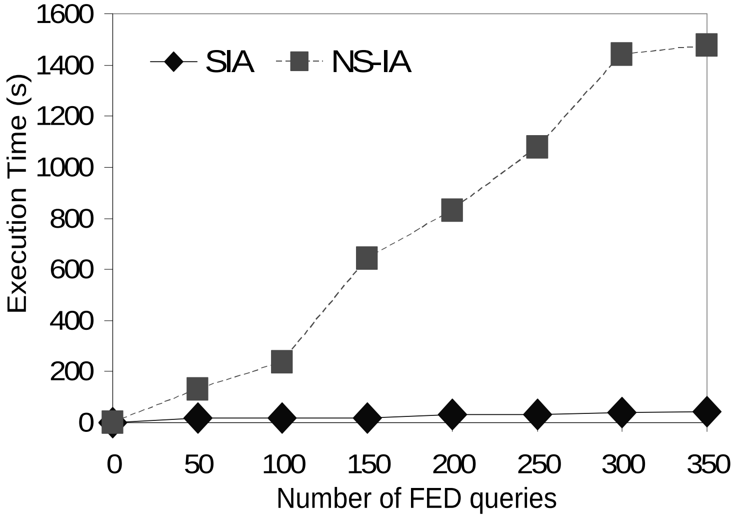


Figure 10.9: Aggregate sharing on FED. Comparing total execution times of shared query network (SIA) and non-shared query network (NS-IA). SIA is up to hundreds-fold faster.

10.3.3 Study on Vertical Expansion

To study how the incremental size $|S_N|$ influences the vertical expansion performance, we select two FED query pairs and compare vertical expanded plans (VE) and non-vertical expanded plans ($NonVE$) on them by varying the incremental sizes from 1 to 30000 tuples in exponential scale, as shown on the X axis in Figures 10.11 and 10.12. In each query pair, one query B can be shared by vertical expansion from the other query A, and their data

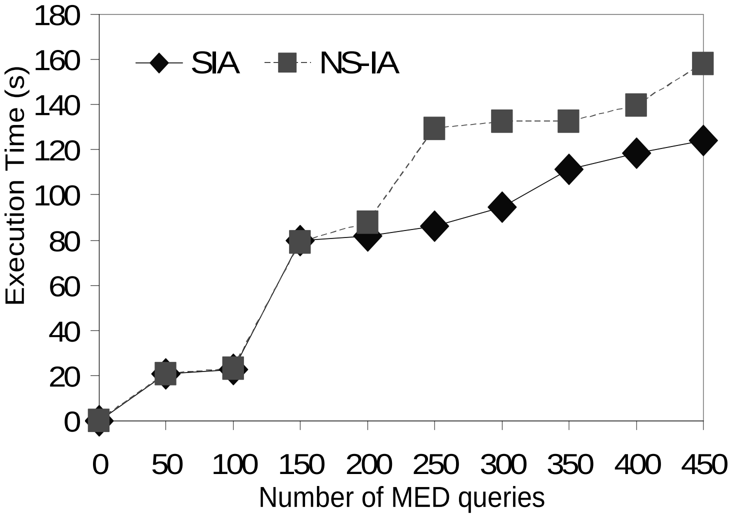


Figure 10.10: Aggregate sharing on MED. Comparing total execution times of shared query network (SIA) and non-shared query network (NS-IA). SIA is faster.

sizes are shown in Table 10.5. Figures 10.11 and 10.12 show two types of execution times, IBT and ITT. IBT is the time to update the whole incremental batch S_N, indicated on the left Y axis, and ITT is the average time to update an individual tuple in each incremental batch, $ITT = IBT/|S_N|$, indicated on the right Y axis. The VE performance gets close to NVE as $|S_N|$ gets larger, since $|S_N|^2$ in T_{curr} becomes dominant and dims the VE advantage; see Section 4.2 for the time complexity discussion. We expect that this situation can be avoided with the hasing implementation.

	Pair1	Pair2
$\|S_H\|$	300000	300000
$\|A_H\|$	95050	94895
$\|B_H\|$	10000	10000

Table 10.5: Vertical expansion statistics

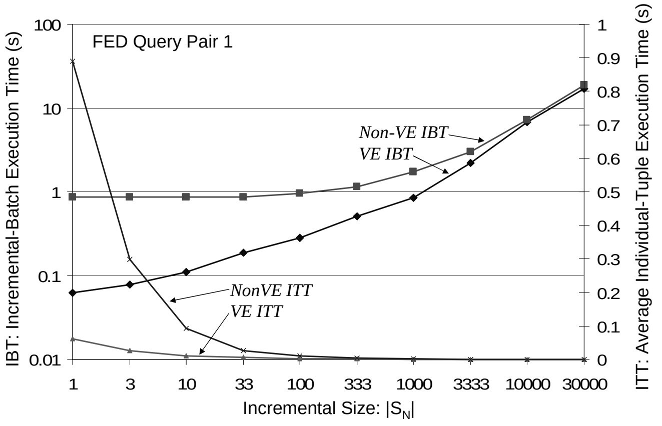


Figure 10.11: Effect of vertical expansion for the first example. VE is always better, but the benefit diminishes as the incremental size increases.

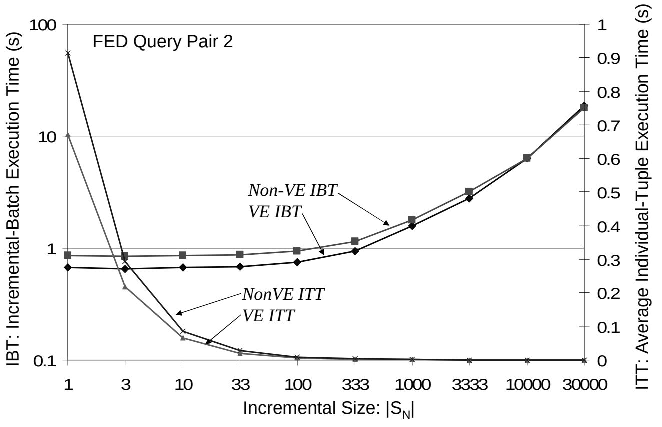


Figure 10.12: Effect of vertical expansion for the second example. VE is always better, but the benefit diminishes as the incremental size increases.

10.4 Incremental Multiple Query Optimization on SJP Queries

In this section, we present the experiments on IMQO on the five SJP query sets, QIMQOSJPFed-Manual, QIMQOSJPFed-Uniform, QIMQOSJPFed-Normal, QIMQOSJPFed-GaussianMixture, and QIMQOSJPMed. The experiment setting is as discussed in Section 10.1. For each query set, the performance is evaluated on increasingly-expanded query subsets with 100 more queries added to the system each time.

We observe that the benefit of IMQO is much more significant on FED than on MED, as for aggregate queries. See Section 10.3.2 for explanations.

We compare performance of four query network generation configurations, AllSharing, NonJoinS, NonCanon, and MatchPlan, as shown in Table 10.6. 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. 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.

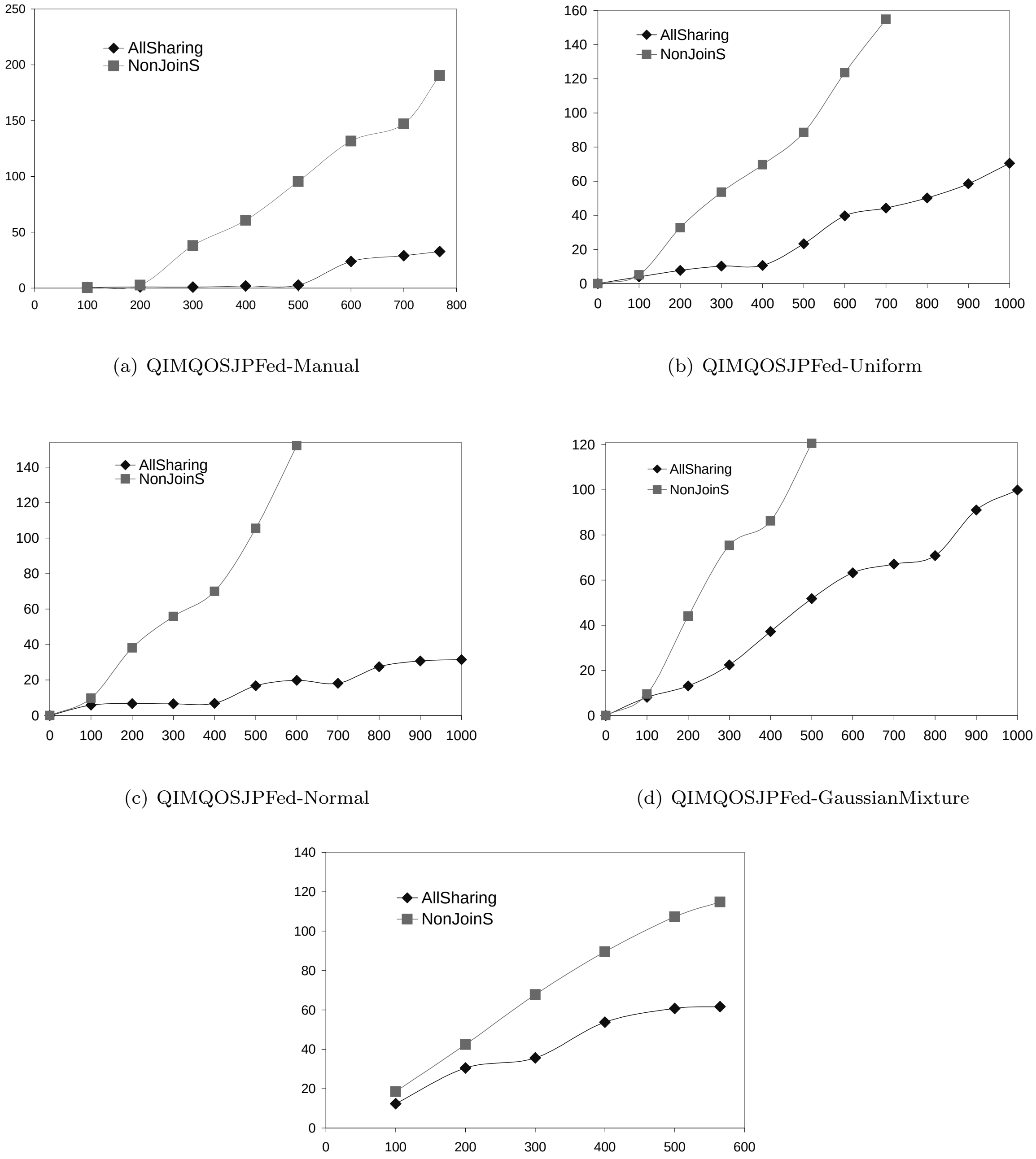
In some of the experiments, when the baseline data points are apparently much worse than the configuration to be compared, they were not evaluated and not shown in the figures. For example, in Figures 10.13(b), 10.13(c), and 10.13(d), NonJoinS performance is reported only up to 700, 600, and 500 queries, respectively.

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

Table 10.6: Network Generation Configurations. Functionality enabled: Y; disabled: N.

10.4.1 Incremental Multiple Query Optimization

As shown in Figure 10.13, 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.



(a) QIMQOSJPFed-Manual

(b) QIMQOSJPFed-Uniform

(c) QIMQOSJPFed-Normal

(d) QIMQOSJPFed-GaussianMixture

(e) QIMQOSJPMed

Figure 10.13: Join Sharing. Comparing total execution times in seconds of join-shared (AllSharing) and non-join-shared query networks (NonJoinS). AllSharing is up to tens-fold faster.

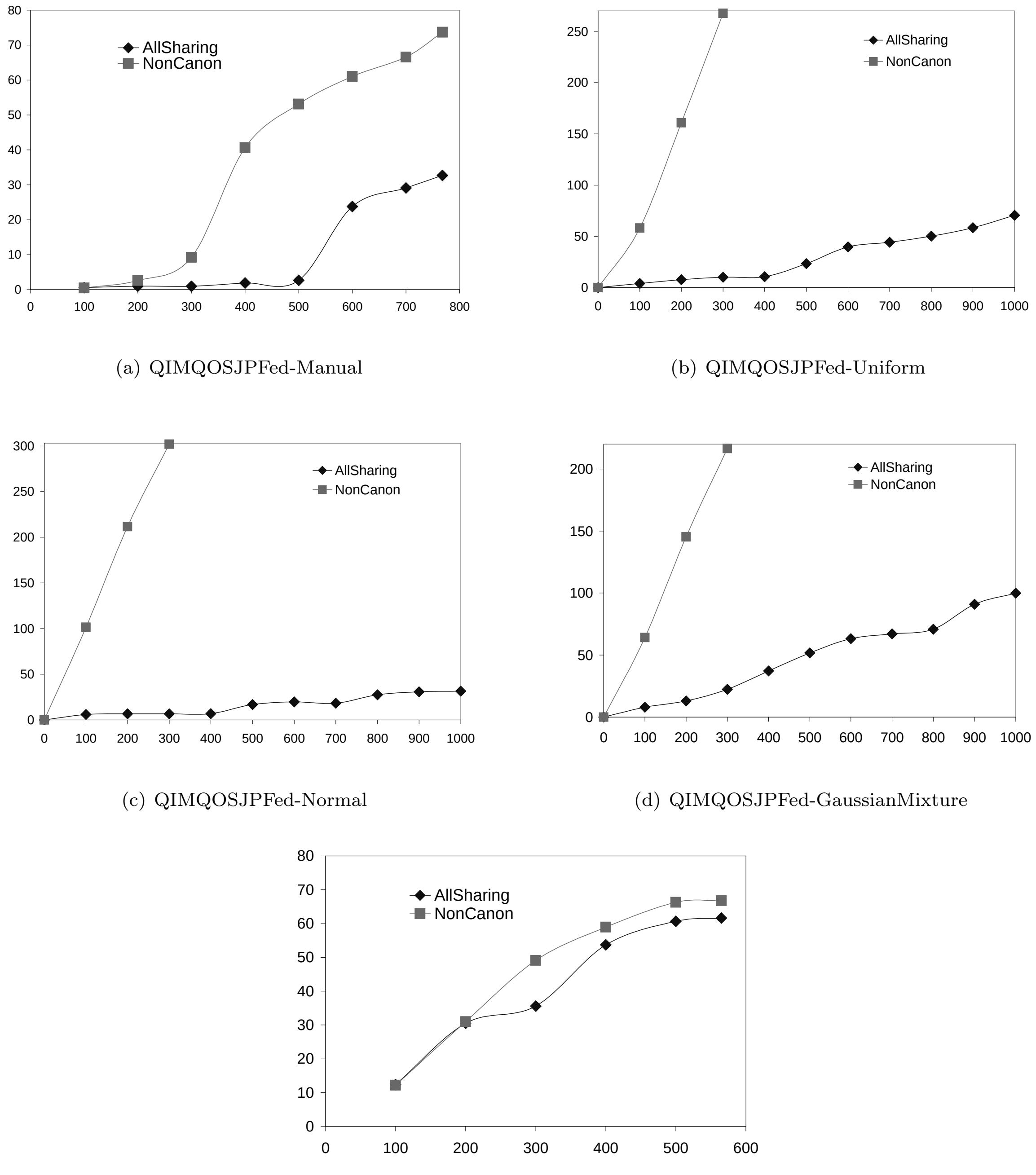
10.4.2 Canonicalization

As shown in Figure 10.14, 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 Example 1.1, 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 subsumption relations between such join predicates. Thus the sharing leads to more significant reduction in the number of join nodes.

We observe that the canonicalization outperforms join sharing on the randomly parameterized query sets, QIMQOSJPFed-Uniform, and QIMQOSJPFed-Normal, QIMQOSJPFed-GaussianMixture, but not on the manually parameterized query sets, QIMQOSJPFed-Manual, and QIMQOSJPMed. This is because that randomly parameterized queries present rare equivalent predicates, but many more subsumption predicates. Without canonicalization, many subsumptions can not be identified. For example, the subsumption between the join predicates $r2.tran\_date \leq r1.tran\_date + 20$ and $r2.tran\_date \leq r1.tran\_date + 10$ can not be identified without canonicalization. Therefore, many potential subsumption join sharing opportunities are not explored.

We observe that different randomly parameterized query sets present different levels of performance improvement by conducting join sharing or canonicalization. Comparing the three randomly parameterized query sets in Figures 10.13 and 10.14, we notice that the most significant improvement comes from QIMQOSJPFed-Normal, the second from QIMQOSJPFed-Uniform, and the last from QIMQOSJPFed-GaussianMixture.

Examining the queries and the constructed query networks, we find that QIMQOSJPFed-Normal presents the highest level of sharability, while QIMQOSJPFed-GaussianMixture presents the lowest level of sharability, because of the following. The subsumption sharing between two queries requires that the parameter values present the consistent predicate subsumption relationship from one query to the other. Remember the parameter values within one query are drawn independently. When the values focus in smaller ranges (values fall into smaller ranges with higher probability), the subsumption is more likely to occur. Clearly, QIMQOSJPFed-Normal generated from normal distributions has the most focused values. As discussed in Section 10.1.3, a Gaussian mixture has two local focuses located at $\mu_1 = min + (max - min) * 0.2$, $\mu_2 = min + (max - min) * 0.8$, and each Gaussian model has the half probability to be chosen and has $\sigma = 0.1 * (max - min)$. This is more scattered than the uniform distribution in range $[min, max]$.



(a) QIMQOSJPFed-Manual

(b) QIMQOSJPFed-Uniform

(c) QIMQOSJPFed-Normal

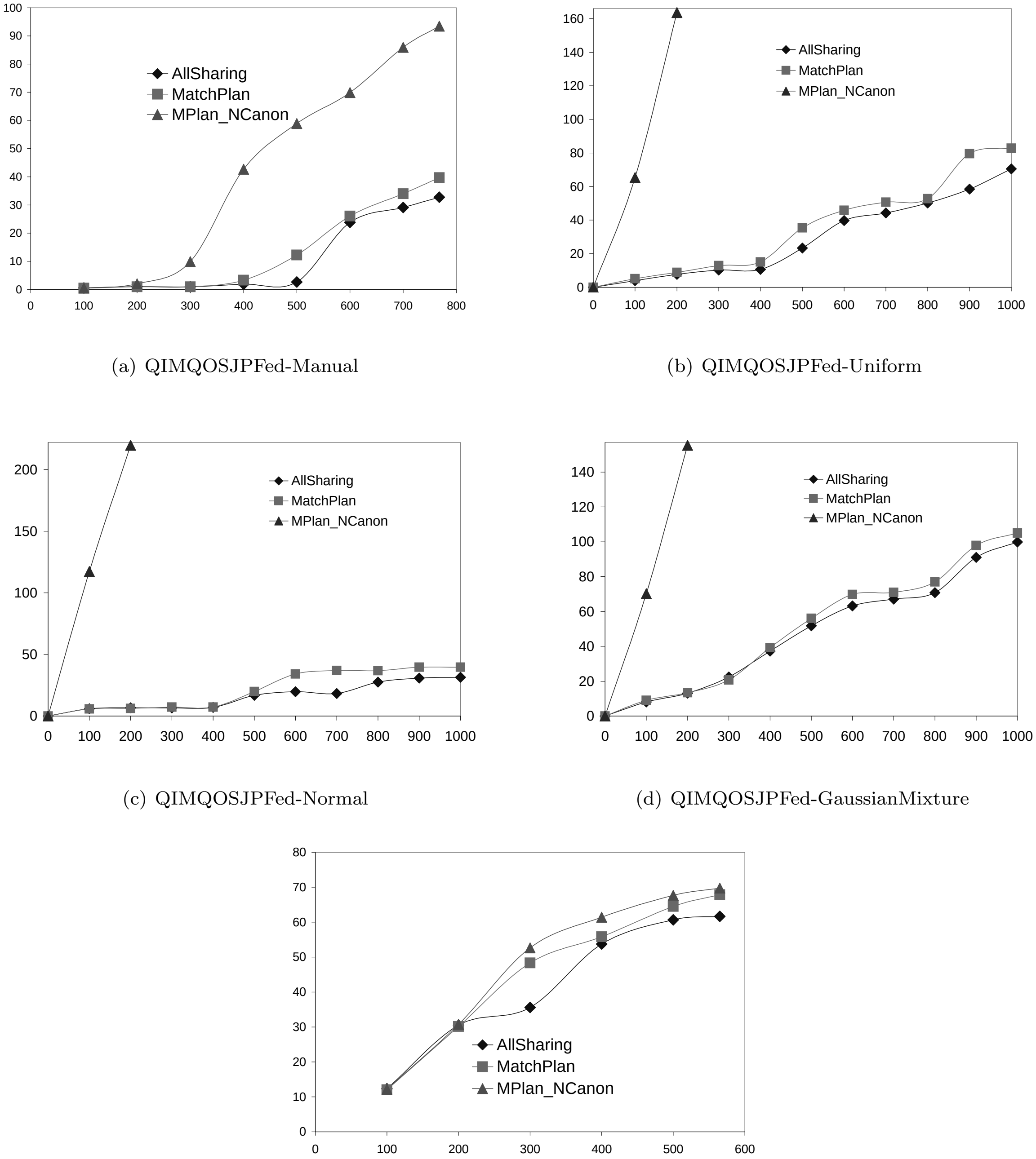
(d) QIMQOSJPFed-GaussianMixture

(e) QIMQOSJPMed

Figure 10.14: Canonicalization. This shows the effectiveness of canonicalization. The canonicalized query networks are up to 50 folds performance improvement.

10.4.3 Match-Plan versus Sharing-Selection

In Figure 10.15, 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.



(a) QIMQOSJPFed-Manual

(b) QIMQOSJPFed-Uniform

(c) QIMQOSJPFed-Normal

(d) QIMQOSJPFed-GaussianMixture

(e) QIMQOSJPMed

Figure 10.15: Match-plan vs. sharing-selection. This compares the total execution times of query networks generated with sharing-selection (AllSharing), match-plan (MatchPlan), and match-plan without canonicalization (MPlan_NCanon, the baseline). AllSharing is the best.

10.4.4 Weighted Query Network Size

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. The weighted size intends to predict the total query network execution time by assigning different weights to nodes of different types (selection or join) at different join depths based on cost estimates and summing-up them together. Particularly, the weight of each node intends to estimate the execution cost of that node. The cost is estimated based on the node's type (selection or join) and join depth. The cost largely depends on the number of records to be processed in an incremental evaluation run. Assuming the queries filter out records through selection predicates and join much less records than the original data, the shallow join-depth nodes should have more weights than the deep ones. Since join is usually much more expensive than selection, join nodes should have much more weights than selection nodes.

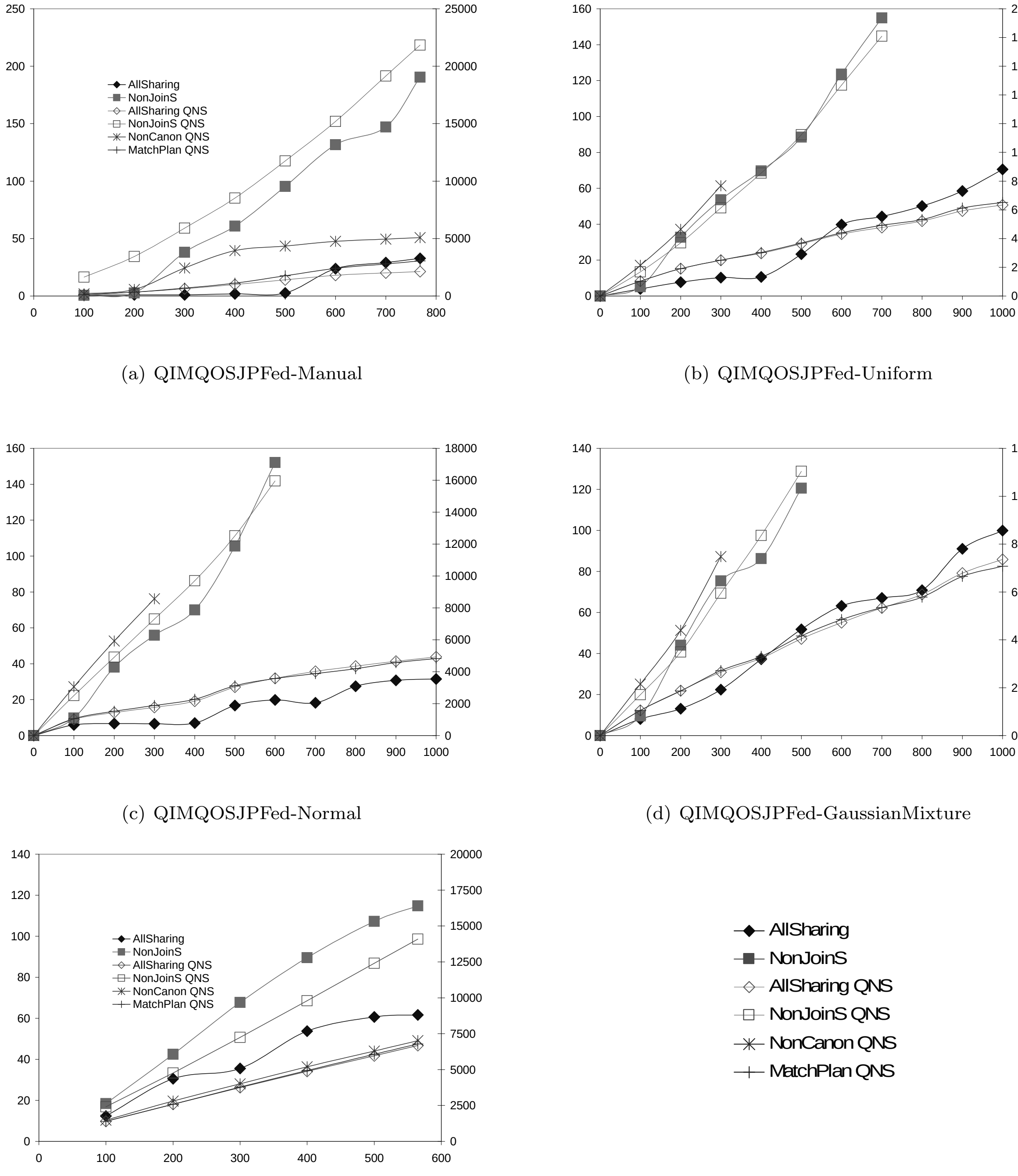
Node	S0	S1	S2	S3	S4	J1	J2	J3	J4
Weight	5	4	3	2	1	16	8	4	2

Table 10.7: Node weights. Node legend: S: selection node; J: join node; 0-4: join depth.

Table 10.7 shows the weights we used for computing the weighted query network sizes. The selection node weights decrease linearly as the join depths increase, and the join node weights decrease exponentially as the join depths increase. The sizes are plotted in Figures 10.16. The figures show that canonicalization and common computation identification are very effective. 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 10.16(a) etc.), we believe that it 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.



(a) QIMQOSJPFed-Manual

(b) QIMQOSJPFed-Uniform

(c) QIMQOSJPFed-Normal

(d) QIMQOSJPFed-GaussianMixture

(e) QIMQOSJPMed

(f) Legend

Figure 10.16: Fed Weighted Query Network Sizes (QNS). QNS curves are consistent to the execution performance.

Chapter 11

Conclusion and Future Work

This concluding chapter discusses the related issues and future work directions, and summarizes the thesis contributions.

We discuss several problems related to multiple continuous query optimization and execution, including long history handling, query deregistering, and batch-size tradeoff. Then we discuss the future work that can further improve the performance in three categories, including the immediate generalization of current work, adding adaptive processing functionalities, and adapting our methods to work well on new computing infrastructures. And finally we summarize the thesis contributions.

11.1 Discussion

11.1.1 Handling Long History

Continuous queries may deal with long or potentially unlimited historical data. To address the problem, researchers have developed techniques to bound the history. The techniques fall in two categories, exploiting constraints and approximation. Streams usually hold certain constraints that can be exploited to drop useless historical data without compromising result precision. Approximation drops stream tuples to limit the history with certain error bound on results. Currently, ARGUS does not have such techniques. In the rest of the section, we briefly discuss the techniques and the challenge of implementing them on shared query networks.

Exploiting Constraints

Streams usually hold certain constraints or display certain data patterns such that historical stream tuples useless to the current set of continuous queries can be dropped. When a new continuous query is registered, it may request more historical data beyond the currently available ones to produce precise results. However, by adjusting the history retaining policy, this problem will be alleviated continuously and eventually go away as new data pour in.

Babu et al. [18] discussed three kinds of stream constraints and how they can be used to reduce history for selection-join-projection (SJP) queries. The constraints are referential integrity constraint, clustered-arrival constraint, and monotonicity constraint. The last two constraints can also be used for aggregate queries. We review the constraints and the applications to SJP queries that were discussed in [18], and add the discussion on handling aggregates and defining minimum sliding window.

Referential integrity constraints on data streams (RIDS) characterize data arrival patterns on two streams that can be joined in the many-one form. RIDS on a many-one join $S_1 \rightarrow S_2$ states that when a tuple s_1 arrives on S_1, its unique joining tuple in S_2 has already arrived [18]. Due to physical reasons, such as network delay, the constraints may not hold strictly. A more relaxed k-constraint version states that when a tuple s_1 arrives on S_1, its joining tuple s_2 has already arrived or s_2 will arrive within k tuple arrivals on S_2 [18]. To be realistic, we assume the query has selection predicates on S_1 and S_2. According to the k-constraint, when a new tuple $s_1 \in S_1$ arrives, if it joins with $s_2 \in S_2$, it will not join with further tuples in S_2 and can be discarded. If s_1 does not join with any tuple in

S_2, s_1 is retained until it joins with a future tuple s_2 within k tuple arrivals or expires at the kth tuple arrival. When a new tuple $s_2 \in S_2$ arrives, if it fails to pass the selection predicates, any tuples in S_1 that joins with it can be discarded. Although then s_2 can be discarded now, it may be retained to fail future joining tuples, so that they don't need to wait until the expiration.

A clustered-arrival constraint (CA) on attribute A of a stream S specifies that duplicate values for A arrive at successive positions in S. The relaxed k-constraint version states that the number of S tuples with non-v values for attribute A between any two S tuples with the same v-value is less or equal to k. Consider the many-one join $S_1 \rightarrow S_2$ where S_1 holds the CA constraint on the join attribute A. Once a tuple $s_2 \in S_2$ joins with some tuple $s_1 \in S_1$ with value $A = v$, s_2 will not join with further S_1 tuples after observing a no-show of $A = v$ in any subsequent $k + 1$ tuple window on S_1. So s_2 can be dropped as well as any tuples joining with s_2.

CA constraints can also be used to drop tuples for aggregate queries. Consider a holistic aggregate query on stream S that groups tuples by values of attribute A which holds the CA constraint. Once a new value $A = v$ is seen from S, we can compute and update the aggregate for this new group until observing a no-show in any $k + 1$ tuple window. Then the v-value tuples can be dropped.

An ordered-arrival constraint (OA) on attribute A of a stream S says that the stream S is sorted on A. The relaxed k-constraint version allows some disorder within k tuple windows. It specified that for any tuple s and any tuple s' that arrives at least $k+1$ tuples after s, $s.A < s'.A$. OA constraints can be used to drop tuples for both many-one and aggregate queries in the similar way as CA constrains. Furthermore, OA constraints allow dangling tuples (tuples never join) to be dropped.

An OA constraint can also be used to define the minimum sliding windows, such as those on the timestamp attribute. Consider a query that concerns only with recent data in a sliding window defined on attribute A from stream S. Assume A is in the ascending order (descending order is symmetric). The sliding window is specified as $Current - S.A < c$, where c is a constant, and $Current$ is the current value of A. $Current$ could be either an objective value independent of the stream, such as the actual time, or the $S.A$ value of the latest tuple. With the relaxed k-constraint version, any tuple not in the window plus the k preceding tuples can be dropped.

Approximation

Various approximation techniques are described in rich literature. Load shedding [2] drops chunks of tuples when the system performance degrades to a certain level. More sophisticated approaches achieve certain error bounds by sampling streams according to statistical models, such as histogram [116] or compressed wavelets [52, 25] for aggregation and join, and Bloom filter [21] for duplicate elimination, set difference, or set intersection.

Adaptive approximation [89] is important to deal with the dynamics of stream processing including fluctuations in data rates and distributions, query workload, and resource availability. For example, besides the reduction on the original stream data, more fine-tuned approximation may be considered at operator level. When an operator becomes the bottleneck for the entire query evaluation plan, a dynamic sampling operator can be applied before the congested operator. Such challenges have not yet been investigated in depth. An interesting direction in this regard is using machine learning techniques to learn the model of the dynamics and using the model to guide the adaptive approximation.

Here we review the approximation techniques in literature on two problems, k-way foreign key joins [3], and distributed top-K or quantile monitoring.

A 2-way foreign key join is a join on referential-integrity-held attributes. It is the many-one join discussed in Section 11.1.1. We use the term foreign key join in this section to be consistent with the discussion in [3].

A k-way foreign key join on k relations (denoted as a set R) is defined as following. A k-way join is a k-way foreign key join if each relation in R has at least one 2-way foreign key join with another relation in R.

We can construct a join graph G for a k-way foreign key join, in which each relation is presented as a node, and each 2-way foreign key join is presented as a directed edge from the foreign key relation (many) to the primary key relation (one). It can be shown that the graph G is a connected directed acyclic graph with a single root node S. Intuitively, S is the central fact relation in a star or snowflake schema. Further, it can be shown that a uniform sample on S will give a uniform sample on the final result.

Now we discuss distributed top-K and quantile monitoring. Aggregates vary on history retainment. Algebraic aggregates on unlimited history or windowed history do not need to retain the entire history, but only a bounded number of up-to-date statistics. Some algebraic aggregates on unlimited history become holistic aggregates on windowed history. Such as MAX, MIN, and top-K queries. Holistic queries, such as quantiles, need to retain the entire history (unlimited or limited) to produce precise results. For the holistic queries, we can apply approximation.

MAX or MIN can be approximated by maintaining the top-K answers instead of the entire history. K is a tunable parameter. Here we use the MAX as the example (MIN is symmetric). When the maximum value tuple expires from the time window, the second value tuple becomes the result. When a new tuple arrives, if the value is larger than that of at least one tuple in the top-K list, the list is updated. Such operation has the effect of monotonic increasing on minimum value in the list. So the list may shrink as old top-K tuples expire while no new tuples can be inserted into the list. A remedy that resumes the history retaining can be evoked when shrinkage is detected. The remedy does not guarantee precise results. An extreme case occurs when the values arrive in the descending order, as shown in Figure 11.1. Here the time window is specified in the number

N of tuples. In the time period between t_1 and t_3, the top-K list is the tuples arrived between t_2 and t_2. Starting from t_3, the top-K list shrinks, the remedy is evoked, and precise results continue to be processed until time t_4. At time t_4, the top-K list is empty, but the retained recent history has only K tuples. Clearly, when $K = N$ the results are always correct, but may not be necessary for tolerable error bounds. How to tune K with regard to N and the data characteristics is an interesting problem for future research.

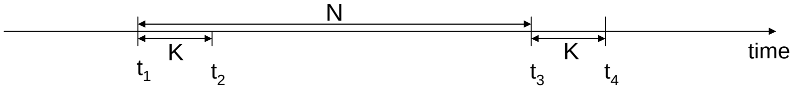


Figure 11.1: N-tuple sliding time window maintaining the top K results.

Babu and Olston [14] described the efficient distributive top-K monitoring, where the aggregate values are computed from distributed nodes, e.g. most popular webpages that can be accessed from mirrored distributed servers. The focus is on producing error-bounded results with minimum communications among nodes. The idea is that after an initial top-K is computed, constraints are established in local nodes. The central node estimate the results based on the constraints. Local nodes monitor the local streams but not transmitting the updated values until the constraints are violated. The constraints are extended by Cormode et al. [38] to data distribution models that are used to estimate distributed quantiles at central station and are monitored for violation by local nodes.

Implementation

In future, ARGUS can be enhanced with constraint exploitation and approximation. We can apply the techniques on original streams. We note that the techniques can be applied to intermediate result streams as well.

Different intermediate result streams display very different data patterns and data rates. Some of them benefit from the techniques while others (low-rate stream, etc.) do not. Further, in a shared query network, different queries may have different priorities (requirement on response time or accuracy, etc.) and incur different computation cost. A challenging question is how to decide, probably dynamically, which original or intermediate streams to use which techniques to optimize a certain utility function, such as optimizing the throughput or response time for high-priority queries.

11.1.2 Deregistering Query

As the inverse of adding new queries into a shared query network, deregistering existing ones from the network also naturally occurs in application. Currently, ARGUS does not take any action when deregistering an existing query. The analyst may simply not poll the produced results.

The system can be enhanced in deregistering query by deleting the nodes that are not useful anymore. The system will check the system catalog to identify the nodes that are used for produce results for the deregistered query and are not shared by other queries. These nodes are dropped from the query network and the system catalog is updated.

As the shared query network shrinks, its topology may drift away from optimality. To deal with this, local reoptimization, such as rerouting, restructuring, and merging, can be applied to regain the local optimality.

More intelligent reconstruction strategies model query dynamics. New queries tend to be revised more often. At the beginning, analysts do not fully understand the phenomena, and tend to either overestimate or underestimate the situations. As time pass by, refined queries stabilize gradually. It could be beneficial to keep the new and unstable queries outside the shared query network. They are individually optimized and executed. When one of them is deregistered, the entire query network can be dropped without interfering the rest of the active queries. When new queries reach the stable status, they can be added into the shared query network.

To automatically adjust the query network sharing, the system must understand the behavior of analysts. This brings up a new and challenge research problem of behavior modeling. The goal is to predict the probability that a query will be revised in a certain period of time. Then the sharing decision can be made based on the value of the probability. The probability may be modeled from several factors including subjective measures given by analysts, the recent revision history, and the result production history (when too many false-alarm results are produced, the query tends to be revised to produce less, etc.).

11.1.3 Immediate Response versus Processing Efficiency

Ideally, whenever a new stream tuple arrives, the shared query network immediately processes it and produces results for all the queries. However, due to the limit of computing power, we often need to trade off between immediate response and processing efficiency. Since each node operator imposes computation overhead, processing tuples in batch reduces the overhead cost and improves the overall performance [2]. Sections 10.2.1 and 10.3.3 demonstrated this effect on SJP and aggregate queries. However, the batch size (the number of tuples in the batch) should not be too large to unnecessarily delay the response time or to invoke too many page swapping.

Finding the optimal batch size is a challenging scheduling problem related to query execution engine. The optimal batch size keeps the system in a busy but non-congested state. Whenever the system completes the processing on one batch, the unprocessed newly-arrived tuples just compose the entire next batch. Due to query and data dynamics, the optimal batch size changes over time. Monitoring the system status and dynamically adjusting the batch size present another problem for adaptive query processing. And it should be considered with approximation.

In general, finding the optimal batch size with appropriate level of approximation is very hard, and involves complex cost modeling. However, in the continuous query processing scenario, it is possible to adopt simple feedback/adaptation mechanism that is widely used in automatic control systems to approach the optimal status. Particularly, when the system is congested or idles, the batch size and the level of approximation increase or decrease a small amount. The amount is determined by a parameterized function of the level of congestion and idleness. The parameters control the level of adjustment in response to the status changes, and should be tuned to minimize the adapting time and avoid vibration (alternative overestimation and underestimation of the batch size and the level of approximation).

A related problem with regard to response time is that some queries require immediate response while the rest do not. To provide immediate response to high-priority queries, system should schedule the execution of their nodes first. Optimal node execution scheduling is discussed in Section 11.2.2.

11.2 Future Work

We look at future work in three categories, generalization of current work, adding new capabilities, particularly adding adaptive processing functionalities, and adapting our methods to work well on new computing infrastructures.

11.2.1 Generalization of Current Work

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

To support multi-way joins, we consider general conditional materialization, in which intermediate results of 2-way joins may be omitted. Since the database is large, and the results may not necessarily be sparse as expected in anomaly detection, the intermediate results of joins may be also large enough such that the materialization cost may offset the time saved from repetitive computation. Whether or not materializing an intermediate join result set can be decided by cost estimation. Performing conditional materialization on joins is equivalent to performing multi-way joins. The challenge is how to index and search multi-way join computations in the IMQO framework. We have preliminary ideas of indexing the information in separate system catalog relations, and will investigate the search algorithms.

Beyond the match-plan and sharing-selection, we propose to investigate more sophisticated local re-optimization techniques including restructure sharing and rerouting. Since the global optimization is NP-complete, the efficient local re-optimization is practical and worth studying. Restructure sharing is a reconstruction of a local network structure. It occurs when a set of new predicates need to be evaluated and the local structure can be reconstructed to provide the results for both the new predicates and the existing predicates more efficiently. Rerouting is changing the direct parent nodes of a set of existing nodes. It occurs after a new node is created and a set of existing nodes can be more efficiently evaluated from the new node.

11.2.2 Adaptive Query Processing

Adaptive query processing (AQP) has received much attention recently in the stream processing research. AQP techniques at various DSMS processing layers are proposed.

They fall into three major categories, dynamic query plan re-optimization [131], adaptive operator scheduling [16], and batch-size adaptation and approximate query answering [2]. The first one is jointly handled by the stream execution engine and the plan generator, and the last two are mainly handled by the engine. In this subsection, we discuss the thoughts in each direction.

While AQP has been studied extensively in all the three directions, applying AQP on IMQO-generated query network has not yet been investigated. An IMQO-enabled system presents both new challenges and opportunities to apply AQP. An IMQO-enabled system tends to support a large number of queries, and identifying the congested local plan regions that need adaptive processing adjustment is challenging. On the other hand, equipped with a comprehensive computation indexing scheme that provides a clear view of the current plan and fast search and update tools, the system can easily apply re-optimization strategies, such as adaptive plan restructuring and rerouting as discussed in Section 11.2.1.

To support dynamic plan re-optimization, the system needs to look at the local plan region, and reconstruct the new local plan based on the new cost estimates. The IMQO-enabled system already provides the basic functionalities, e.g. plan presentation and searching tools, to support such operations. And the extensions to support re-optimization, such as restructuring and rerouting, will provide high-level functional modules to realize dynamic and adaptive re-optimization. However, successful low-cost and effective adaptation also relies on accurate pinpointing major congestions on the fly and quick plan switch. These two problems should be addressed at the execution engine side, and remain part of the research.

Dynamic operator scheduling is another important adaptive processing technique. Query network nodes are evaluated sequentially. The evaluation order only needs to satisfy one constraint: a node must be evaluated before any of its descendants. Thus when there are branches in the query network, there are more than one valid evaluation order. Our current system takes a breadth-first traversal order. Our experiments and analysis show that different orders on a large network may lead to big performance differences due to different page swapping patterns. Since a large network can not fit into memory, the query engine has to perform page swapping operations between the memory and disk if the part of the network to be processed is not in the memory. In the best situation, every part of the network is loaded into memory just once and is prefetched into memory before its turn for processing. Then the overhead of page swapping is just the loading time of the first part. In the worst situation, every part has to be loaded into memory when it is requested. Thus the overhead of page swapping is the sum of the loading time of all parts of the network. In practice, if we evaluate a set of nodes that access the same part of network contiguously, then that part of the network does not need to go through many page swapping. However, it is not always possible to follow such requirement restrictly. For example, a node table T is accessed by writing when it is evaluated, and is accessed by reading when its direct descendant nodes are evaluated. If T has sibling nodes that share the same direct parent, and each sibling node have its own descendants. Then T may not be put contiguously with its own descendants. For another example, a join node has to be evaluated after both of its direct parent nodes are evaluated. Thus finding an optimal evaluation order to minimize the page swapping cost is not a trivial task. Furthermore, the node execution order should also provide faster responses to high-priority queries. However, by modeling page swapping costs on network nodes which depends on their table sizes and the local topology, and quantify the query priority as costs, we turn such optimization into a search problem. The solution, possibly with heuristics, will give us an optimal order to evaluate the large query network.

Finally, adaptively adjusting batch-size and the level of approximation is important for providing fast response time and avoiding overloading. As discussed in Section 11.1.3, finding the optimal batch-size with appropriate level of approximation is very hard, but may be solved by a simple feedback/adaptation mechanism. A parameterized function of the system status (the level of congestion or idleness) is used to control the adjusting amount. Furthermore, the function may be learned dynamically as the system keeps on running. For example, data-rates may reach peaks at certain hours each day, then the optimal adjustment function can be learned from the past performance, and may be adjusted as the peak hours or peak volumes shift. For another example, unexpected data bursts may display similar distribution characteristics, such as similar data-rate changes or time-window width. These data bursts provide the training data to learn the optimal adjustment function that can be applied when the first sign of future unexpected bursts is detected.

11.2.3 New Infrastructures

Parallel/Distributed computing is the natural way to go for big performance improvement. While parallel/distributed computing algorithms may find ways to parallelize/distribute evaluation work at lower level, it is clearly more advantageous to use query network cost models to obtain optimal workload partitions. The problem is similar and coupled to that of finding the optimal evaluation order. However, the cost model should be different. It should also consider communication traffic and bandwidth, and avoiding bottlenecks, etc. We expect to develop a set of algorithms to compute the optimal workload partition and assignments to a given parallel/distributed computing setting. If possible, we will experiment such algorithms on some parallel/distributed platforms.

Another direction is exploring automatic index selection and related techniques. Since indices on certain distributed tables can speed up certain types of queries (range queries, etc.) with non-negligible maintenance overhead, automatically selecting proper indices for query network tables is a challenge but is worth investigating. Automatic index selection has been studied for DBMS, but not studied for large-scale connected query networks. Therefore, we will be able to start with experimenting with existing methods, and we expect extensions to be developed to suit for our particular purpose.

11.3 Summary of Contributions

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

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

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

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

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

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

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

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

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

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

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

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

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

[3] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *SIGMOD Conference*, pages 275–286, 1999.

[4] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[5] Sameet Agarwal and et al. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

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

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

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

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

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

[11] Ahmed M. Ayad and Jeffrey F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of data*, pages 419–430, Paris, France, June, 2004.

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

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

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

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

[16] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In Weikum et al. [123], pages 407–418.

[17] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of data*, pages 407–418, Paris, France, June, 2004.

[18] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting -constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

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

[20] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[21] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

[23] Jaime Carbonell, Phile Hayes, Cenk Gazen, Chun Jin, Aaron Goldstein, Ganesh Mani, and Johny Mathew. Finding Novel Information in Large, Constantly Incrementing Collections of Structured Data. In *NIMD PI Meeting*, San Diego, CA, 2003.

[24] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[25] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. In *VLDB*, pages 111–122, 2000.

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

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

[28] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, January, 2003.

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

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

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

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

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

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

[35] Zhimin Chen and Vivek R. Narasayya. Efficient computation of multiple group by queries. In *SIGMOD Conference*, pages 263–274, 2005.

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

[37] Graham Cormode and et al. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD Conference*, pages 25–36, 2005.

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

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

[40] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[41] David DeHaan, Per-Åke Larson, and Jingren Zhou. Stacked indexed views in Microsoft SQL Server. In *SIGMOD Conference*, pages 179–190, 2005.

[42] Amol Deshpande, Suman Nath, Phillip B. Gibbons, and Srinivasan Seshan. Cache-and-Query for Wide Area Sensor Databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 503–514, San Diego, California, June, 2003.

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

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

[45] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.

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

[47] Eugene Fink, Aaron Goldstein, Philip Hayes, and Jaime Carbonell. Search for Approximate Matches in Large Databases. In *Proc. of the 2004 IEEE Intl. Conf. on Systems, Man, and Cybernetics.*

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

[49] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148, 1998.

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

[51] Hector Garcia-Molina, Wilburt Labio, and Jun Yang. Expiring data in a warehouse. In *VLDB*, pages 500–511, 1998.

[52] Minos N. Garofalakis and Phillip B. Gibbons. Wavelet synopses with error guarantees. In *SIGMOD Conference*, pages 476–487, 2002.

[53] Cenk Gazen, Jaime Carbonell, and Phil Hayes. Novelty Detection in Data Streams: A Small Step Towards Anticipating Strategic Surprise. In *NIMD PI Meeting*, Washington, DC, 2005.

[54] Phillip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. Irisnet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.

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

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

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

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

[59] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid 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.

[60] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.

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

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

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

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

[65] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, M. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, R. Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

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

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

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

[69] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

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

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

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

[73] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive Query Processing for Internet Applications. *Bulletin of the IEEE*

Computer Society Technical Committee on Data Engineering, 23(2):19–26, June, 2000.

[74] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, pages 113–124, 1995.

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

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

[77] Chun Jin, Jaime Carbonell, and Philip 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.

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

[79] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

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

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

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

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

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

[85] Amit Manjhi, Suman Nath, and Phillip B. Gibbons. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *SIGMOD Conference*, pages 287–298, 2005.

[86] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, pages 767–778. IEEE Computer Society, 2005.

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

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

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

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

[91] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD Conference*, pages 563–574, 2003.

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

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

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

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

[96] ARDA NIMD Program. www.ic-arda.org/Novel_Intelligence/.

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

[98] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

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

[100] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.

[101] Nick Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

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

[103] Gerald Salton, editor. *Automatic text processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[104] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products. In *PODS*, pages 238–248, 1997.

[105] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of*

17th International Conference on Very Large Data Bases, pages 469–478, Barcelona, Spain, September, 1991.

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

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

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

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

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

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

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

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

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

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

[116] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *SIGMOD Conference*, pages 428–439, 2002.

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

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

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

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

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

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

[123] Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.

[124] Gerhard Weikum, Axel Moenkeberg, Chrstof Hasse, and Peter Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable

Engineering. In *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

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

[126] Jinxi Xu and W. Bruce Croft. Corpus-based stemming using cooccurrence of word cariants. *ACM Trans. Inf. Syst.*, 16(1):61–81, 1998.

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

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

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

[130] Minghua Zhang, Ben Kao, David Wai-Lok Cheung, and Kevin Yip. Mining periodic patterns with gap requirement from sequences. In *SIGMOD Conference*, 2005.

[131] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In Weikum et al. [123], pages 431–442.

Appendix A

Stream Schemas

We use the synthesized FedWire money transfer data (Fed) and the real Massachusetts hospital patient admission and discharge record database (Med) for system testing and evaluation. Each database contains one single data stream. Fed contains a stream of money transfer transaction records, one record per transaction. And Med contains a stream of in-hospital patient discharge records.

A relevant subset of the attributes of the Fed stream is shown below

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

A relevant subset of the attributes of the Med stream is shown blow

RECID	NUMBER(10),	– record ID, the primary key
PROVIDERID	CHAR(11),	– provider control ID
DISCHARGEID	CHAR(11),	– discharge ID
HOSPID	CHAR(4),	– hospital ID
SEX	CHAR(1),	– patient gender
AGE	NUMBER(3),	– patient age
PAYOR	CHAR(1),	– primary payor type
BDATE	DATE,	– date of birth
ADATE	DATE,	– admit date
DDATE	DATE,	– discharge date
ZIP	CHAR(5),	– patient ZIP code
LOS	NUMBER(5),	– gross length of stay in days
DISP	CHAR(2),	– disposition status at discharge
RACE	CHAR(1),	– patient race
DXS_01	CHAR(5),	– principle diagnosis
DXS_02	CHAR(5),	– secondary diagnosis #1
DXS_03	CHAR(5),	– secondary diagnosis #2
DXS_04	CHAR(5),	– secondary diagnosis #3
DXS_05	CHAR(5),	– secondary diagnosis #4
OPS_01	CHAR(4),	– principle procedure
OPS_02	CHAR(4),	– secondary procedure #1
OPS_03	CHAR(4),	– secondary procedure #2
OPS_04	CHAR(4),	– secondary procedure #3
OPS_05	CHAR(4),	– secondary procedure #4
PAYORG	CHAR(1),	– patient origin indicator
TOWNID	CHAR(3),	– town code ID
BYEAR	DATE,	– year of birth
BYM	DATE,	– year/month of birth

Appendix B

Query Examples

Following are queries on FED. Q_1 has also been shown in Section 1.5.

Example B.1 (Q_1) *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:*

```
SELECT  r1.sbank_aba sbank, r1.orig_account saccount,
        r1.rbank_aba rbank, r1.benef_account raccount,
        r1.amount ramount, r1.tran_date rdate,
        r2.rbank_aba ibank, r2.benef_account iaccount,
        r2.amount iamount, r2.tran_date idate,
        r3.rbank_aba frbank, r3.benef_account fraccount,
        r3.amount framount, r3.tran_date frdate
FROM    FedWireTrans r1,
        FedWireTrans r2,
        FedWireTrans r3
WHERE   r2.type_code = 1000                       —p1
 AND    r3.type_code = 1000                       —p2
 AND    r1.type_code = 1000                       —p3
 AND    r1.amount > 1000000                       —p4
 AND    r1.rbank_aba = r2.sbank_aba               —p5
 AND    r1.benef_account = r2.orig_account        —p6
 AND    r2.amount > 0.5 * r1.amount               —p7
 AND    r1.tran_date <= r2.tran_date              —p8
 AND    r2.tran_date <= r1.tran_date + 20         —p9
 AND    r2.rbank_aba = r3.sbank_aba               —p10
 AND    r2.benef_account = r3.orig_account        —p11
 AND    r2.amount = r3.amount                     —p12
 AND    r2.tran_date <= r3.tran_date              —p13
 AND    r3.tran_date <= r2.tran_date + 20;        —p14
```

Example B.2 **(Q_2)** *The analyst is interested if there exists a bank, which received an incoming transaction over 1,000,000 dollars and has performed an outgoing transaction over 500,000 dollars on the same day. The query can be formulated as:*

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

Example B.3 (Q_3) *For every big transaction, the analyst wants to check if the money stayed in the bank or left it within ten days. The query can be formulated as:*

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

Example B.4 (Q_4) *For every big transaction, the analyst again wants to check if the money stayed in the bank or left it within ten days. However he suspects that the receiver of the transaction would not send the whole sum further at once, but would rather split it into several smaller transactions. The following query generates an alarm whenever the receiver of a large transaction (over $1,000,000) transfers at least half of the money further within ten days of this transaction. The query can be formulated as:*

```
SELECT      r.tranid tranid, r.rbank_aba rbank_aba,
            r.benef_account benef_account,
            AVG(r.amount) ramount,
            SUM(s.amount) samount
FROM        FedWireTrans r, FedWireTrans s
WHERE       r.rbank_aba = s.sbank_aba                 AND
            r.benef_account = s.orig_account          AND
            r.tran_date <= s.tran_date                AND
            s.tran_date <= r.tran_date + 10           AND
            r.amount > 1000000
GROUP BY    r.tranid, r.rbank_aba, r.benef_account
HAVING      SUM(s.amount) > AVG(r.amount) * 0.5;
```

Example B.5 (Q_5) *Check whether there is a bank, having incoming transactions for more than $100,000,000 and outgoing transactions for more than $50,000,000 on one particular day. The formulated query is composed of three SQL statements. First two are view definitions, and the last one is the query on the views.*

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

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

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

Example B.6 **(Q_6)** *Get the transactions of Citibank and Fleet on a particular period of time. The query can be formulated as:*

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

Example B.7 **(Q_7)** *The analyst is interested whether Citibank has conducted a transaction on Nov.27, 2002 with the amount exeeding 1,000,000 dollars. The query can be formulated as:*

```
SELECT  r1.tranid tranid,
        r1.sbank_name sbank_name,
        r1.tran_date tran_date,
        r1.amount amount
FROM    FedWireTrans r1
WHERE   r1.sbank_name =' Citibank (New York State)'             AND
        to_char(r1.tran_date, 'YYYYMMDD') =' 20021127'          AND
        r1.amount > 1000000;
```

Example B.8 **(**QA_1**)** *Monitoring account categories whose daily received money is above 10000000.*

```
SELECT     t1.rbank_aba rbank_aba, SUBSTR(t1.benef_account, 0, 1) benef_account,
           TO_CHAR(t1.tran_date, 'YYYYMMDD') rtran_date,
           SUM(t1.amount) ramount, COUNT(*) rcount
FROM       FedWireTrans t1
GROUP BY   t1.rbank_aba, SUBSTR(t1.benef_account, 0, 1),
           TO_CHAR(t1.tran_date, 'YYYYMMDD')
HAVING     (SUM(t1.amount) > 10000000)
```

Example B.9 **(**QA_2**)** *Monitoring banks whose daily received money is above 10000000.*

```
SELECT     t1.rbank_aba rbank_aba,
           TO_CHAR(t1.tran_date, 'YYYYMMDD') rtran_date,
           SUM(t1.amount) ramount, COUNT(*) rcount
FROM       FedWireTrans t1
GROUP BY   t1.rbank_aba, TO_CHAR(t1.tran_date, 'YYYYMMDD')
HAVING     (SUM(t1.amount) > 10000000)
```

Example B.10 (QA_3) *Monitoring account categories whose daily sent-out money is above 10000000.*

```
SELECT     t1.sbank_aba sbank_aba, SUBSTR(t1.orig_account, 0, 1) orig_account,
           TO_CHAR(t1.tran_date, 'YYYYMMDD') stran_date,
           SUM(t1.amount) samount, COUNT(*) scount
FROM       FedWireTrans t1
GROUP BY   t1.sbank_aba, SUBSTR(t1.orig_account, 0, 1),
           TO_CHAR(t1.tran_date, 'YYYYMMDD')
HAVING     (SUM(t1.amount) > 10000000)
```

Example B.11 (QA_4) *Monitoring banks whose daily sent-out money is above 10000000.*

```
SELECT     t1.sbank_aba sbank_aba,
           TO_CHAR(t1.tran_date, 'YYYYMMDD') stran_date,
           SUM(t1.amount) samount, COUNT(*) scount
FROM       FedWireTrans t1
GROUP BY   t1.sbank_aba, TO_CHAR(t1.tran_date, 'YYYYMMDD')
HAVING     (SUM(t1.amount) > 10000000)
```

Following queries are formulated on an imaginary patient admission stream.

Example B.12 **(**A**)** *Monitoring the number of visits and the average charging fees on each disease category in a hospital everyday.*

$$\begin{array}{ll}
\mathit{SELECT} & \mathit{dis\_cat}, \mathit{hospital}, \mathit{vdate}, \\
 & COUNT(*), AVERAGE(fee) \\
\mathit{FROM} & Med \\
\mathit{GROUP\ BY} & CAT(\mathit{disease})\ \mathit{AS\ dis\_cat} \\
 & \mathit{hospital}, \\
 & DAY(\mathit{visit\_date})\ \mathit{AS\ vdate}
\end{array}$$

Example B.13 **(**B**)** *Monitoring the number of visits and the average charging fees in a hospital everyday.*

$$\begin{array}{ll}
\mathit{SELECT} & \mathit{hospital}, \mathit{vdate}, \\
 & AVERAGE(fee) \\
\mathit{FROM} & Med \\
\mathit{GROUP\ BY} & \mathit{hospital}, \\
 & DAY(\mathit{visit\_date})\ \mathit{AS\ vdate}
\end{array}$$

On the MED database, we have 8 query categories. Three monitor potential outbreaks of contagious diseases (QM_1-QM_3), and the others monitor different types of patient histories (QM_4-QM_8). Since patient IDs are anonymized, we use a set of demographical attributes (approximate ID) to approximate the patient IDs, and use a set of predicates (ID predicates) to approximately identify patients from different records. The approximate ID and ID predicates are shown in full in query QM_4, and are abbreviated to ID and $t1.ID = t2.ID$ repectively in the remaining queries.

Example B.14 (QM_1) *Generate an alarm when there is a distinct anthrax occurrence. A distinct anthrax occurrence is defined as an anthrax patient admission beyond the time window of 40 days from any previous anthrax patient admissions. The alarm condition can be formulated as a set difference query that filters out the occurrences that fall into the time window.*

```
SELECT     t1.dxs_01, t2.zip, t2.adate, t2.recid
FROM       MHDCFY01A t1, MHDCFY01A t2
WHERE      t2.adate > t1.adate
  AND      t1.dxs_01 = t2.dxs_01
  AND      t1.dxs_01 >=' 022'
  AND      t1.dxs_01 <' 023'
GROUP BY   t1.dxs_01, t2.zip, t2.adate, t2.recid
MINUS
SELECT     t1.dxs_01, t2.zip, t2.adate, t2.recid
FROM       MHDCFY01A t1, MHDCFY01A t2
WHERE      t2.adate > t1.adate
  AND      t1.dxs_01 = t2.dxs_01
  AND      t1.dxs_01 >=' 022'
  AND      t1.dxs_01 <' 023'
  AND      t2.adate < t1.adate + 40
GROUP BY   t1.dxs_01, t2.zip, t2.adate, t2.recid
```

Example B.15 (QM_2) *Generate an alarm when there is an anthrax occurrence that is within 40-day time window of the closest previous occurrence but is beyond the 50-mile radius of that one.*

$$\begin{array}{ll}
\textit{SELECT} & t1.dxs\_01,\ t2.zip,\ t2.adate,\ t2.recid \\
\textit{FROM} & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
\textit{WHERE} & t2.adate > t1.adate \\
\quad AND & t1.dxs\_01 = t2.dxs\_01 \\
\quad AND & t1.dxs\_01 >=' 022' \\
\quad AND & t1.dxs\_01 <' 023' \\
\quad AND & t2.adate < t1.adate + 40 \\
\textit{GROUP BY} & t1.dxs\_01,\ t2.zip,\ t2.adate,\ t2.recid \\
\textit{HAVING} & min(zipdistance(t2.zip,\ t1.zip)) >= 50
\end{array}$$

Example B.16 (QM_3) *Generate an alarm when there is an anthrax occurrence that is within 40-day time window of the closest previous occurrence and is within the 50-mile radius of that one.*

$$\begin{array}{ll}
\textit{SELECT} & t1.dxs\_01,\ t2.zip,\ t2.adate,\ t2.recid \\
\textit{FROM} & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
\textit{WHERE} & t2.adate > t1.adate \\
\quad AND & t1.dxs\_01 = t2.dxs\_01 \\
\quad AND & t1.dxs\_01 >=' 022' \\
\quad AND & t1.dxs\_01 <' 023' \\
\quad AND & t2.adate < t1.adate + 40 \\
\textit{GROUP BY} & t1.dxs\_01,\ t2.zip,\ t2.adate,\ t2.recid \\
\textit{HAVING} & min(zipdistance(t2.zip,\ t1.zip)) < 50
\end{array}$$

Example B.17 (QM_4) *Find the children less than 10-years-old who developed lung-related (disease code '4800'-'5200') diseases twice within 40 days.*

```
SELECT  t1.bdate, t1.bym, t1.byear, t1.sex, t1.race, t1.zip, t1.townid, t1.payor, t1.age,
        t1.recid, t1.hospid, t1.adate, t1.ddate, t1.los, t1.disp, t1.dxs_01
FROM    MHDCFY01A t1, MHDCFY01A t2
WHERE   (t1.bdate = t2.bdate OR
         ((t1.bdate IS NULL OR t2.bdate IS NULL) AND t1.bym = t2.bym) OR
         ((t1.bdate IS NULL OR t2.bdate IS NULL)
          AND (t1.bym IS NULL OR t2.bym IS NULL)
          AND (t1.byear = t2.byear)))
  AND    t1.sex = t2.sex
  AND    (t1.race = t2.race OR t1.race IS NULL OR t2.race IS NULL)
  AND   t1.zip = t2.zip
  AND   t1.townid = t2.townid
  AND   (t1.patorg = t2.patorg OR
         (t1.patorg =' 9' AND t2.patorg =' 1')
          OR t1.patorg =' 7' OR t2.patorg =' 7')
  AND   t1.age < 10 AND t2.age < 11
  AND   t1.adate <= t2.adate AND t2.adate < t1.adate + 40
  AND   t1.recid! = t2.recid
  AND   t1.dxs_01 = t2.dxs_01
  AND   t1.dxs_01 >=' 4800' AND t1.dxs_01 <' 5200'
```

Example B.18 **(QM_5)** *Find the patients who took bypass operations (operation code category '361') twice within one year.*

$$\begin{array}{ll}
SELECT & t1.ID,\ t1.ops\_01 \\
FROM & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
WHERE & t1.ID = t2.ID \\
\ \ AND & t1.adate <= t2.adate\ AND\ t2.adate < t1.adate + 365 \\
\ \ AND & t1.recid! = t2.recid \\
\ \ AND & ((t1.ops\_01 >=' 361'\ AND\ t1.ops\_01 <' 362')\ OR \\
 & \ (t1.ops\_02 >=' 361'\ AND\ t1.ops\_02 <' 362')) \\
\ \ AND & ((t2.ops\_01 >=' 361'\ AND\ t2.ops\_01 <' 362')\ OR \\
 & \ (t2.ops\_02 >=' 361' AND t2.ops\_02 <' 362'))
\end{array}$$

Example B.19 **(QM_6)** *Find the patients who took kidney transplantation (operation code category '556'), then developed complications.*

$$\begin{array}{ll}
SELECT & t1.ID,\ t1.ops\_01,\ t2.dxs_01 \\
FROM & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
WHERE & t1.ID = t2.ID \\
\ \ AND & t1.ops\_01 >=' 556'\ AND\ t1.ops\_01 <' 557' \\
\ \ AND & (t2.dxs\_01 =' 99681'\ OR\ t2.dxs\_02 =' 99681'OR \\
 & \ t2.dxs\_03 =' 99681'\ OR\ t2.dxs\_04 =' 99681') \\
\ \ AND & t1.adate <= t2.adate\ AND\ t2.adate < t1.adate + 40 \\
\ \ AND & t1.recid! = t2.recid
\end{array}$$

Example B.20 **(QM_7)** *Find the patients who had liver diseases (categories: '070' and '570'-'573') and developed liver cancer ('155') later.*

$$\begin{array}{ll}
SELECT & t1.ID,\ t1.dxs\_01,\ t2.dxs_01 \\
FROM & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
WHERE & t1.ID = t2.ID \\
\ \ AND & t1.adate < t2.adate \\
\ \ AND & (t1.dxs\_01 >=' 070'\ AND\ t1.dxs\_01 <' 071'\ OR \\
 & \ t1.dxs\_01 >=' 570'\ AND\ t1.dxs\_01 <' 574') \\
\ \ AND & t1.disp! =' 20' \\
\ \ AND & (t2.dxs\_01 >=' 155'\ AND\ t2.dxs\_01 <' 156')
\end{array}$$

Example B.21 (QM_8) *Find the patients who transferred (dispose status is transfer, '02') from one hospital to another.*

$$\begin{array}{ll}
\textit{SELECT} & t1.ID,\ t1.hospid,\ t2.hospid \\
\textit{FROM} & MHDCFY01A\ t1,\ MHDCFY01A\ t2 \\
\textit{WHERE} & t1.ID = t2.ID \\
\quad AND & t1.adate <= t2.adate \\
\quad AND & t1.ddate = t2.adate \\
\quad AND & t1.recid! = t2.recid \\
\quad AND & (t1.disp =' 02') \\
\quad AND & t1.hospid! = t2.hospid \\
\quad AND & t1.dxs\_01 = t2.dxs\_01
\end{array}$$

Appendix C

Experiment Results in Numbers

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Rete Data1	3	1	1	16	14	1	1
SQL Data1	45	14	20	20	14	6	6
Rete Data2	3	1	1	16	15	1	1
SQL Data2	31	12	19	19	17	7	6

Table C.1: Execution times of Q1-Q7 in seconds for Figure 10.1. This shows that incremental evaluation (Rete) is much faster than the naive approach (SQL) for the majority of queries (Q1, Q2, Q3, Q6, and Q7) on both data conditions.

	Q1 Data1	Q1 Data2	Q3 Data1	Q3 Data2
Rete TI	3	3	1	1
Rete Non-TI	44	27	18	17
SQL Non-TI	45	31	20	19
SQL TI	20	12		

Table C.2: Execution times in seconds to show the effect of transitivity inference, shown in Figure 10.3. This shows that transitivity inference leads to significant improvements to both Rete and SQL. "Rete TI": Rete generated with transitivity inference. It achieves 20-fold improvement comparing to Rete Non-TI and SQL Non-TI. "Rete Non-TI": Rete without transitivity inference. "SQL Non-TI": original SQL query. "SQL TI": original SQL query with hidden conditions manually added.

	Data1	Data2
Conditional	38	17
Non-Conditional	44	27
SQL	45	31

Table C.3: Execution times in seconds to show the effect of conditional materialization, shown in Figure 10.5. Comparing the execution times of conditional materialization, non-conditional materialization, and running the original SQL, for Q1 on Data1 and Data2.

# of queries	SIA	NS-IA
50	15.93	131.219
100	16.2035	237.586
150	15.9445	641.783
200	32.2495	831.7605
250	32.581	1076.437
300	39.6875	1440.344
350	42.025	1477.453

Table C.4: Execution times in seconds to show aggregate sharing on FED, shown in Figure 10.9. Comparing total execution times of shared query network (SIA) and non-shared query network (NS-IA). SIA is up to hundreds-fold faster.

# of queries	SIA	NS-IA
50	20.5715	20.7185
100	22.743	23.31872
150	79.7855	79.01062
200	81.6715	87.923
250	86.2245	129.141
300	94.716	132.42
350	111.4765	132.42
400	118.345	139.795
450	124.253	158.049

Table C.5: Execution times in seconds to show aggregate sharing on MED, shown in Figure 10.10. Comparing total execution times of shared query network (SIA) and non-shared query network (NS-IA). SIA is faster.

# of queries	NonJoinS	Mplan_NCanon	NonCanon	MatchPlan	AllSharing
100	0.5155	0.492	0.4765	0.453	0.5545
200	2.6875	1.9845	2.602	0.922	0.9765
300	38.008	9.867	9.2185	0.922	0.961
400	60.774	42.672	40.5935	3.266	1.8825
500	95.3205	58.883	53.1175	12.242	2.633
600	131.687	69.8985	61.0315	26.0625	23.8125
700	147.063	85.922	66.594	33.969	29.0935
768	190.4925	93.461	73.711	39.672	32.7185

Table C.6: Execution times on FED query networks, shown in Figures 10.13(a), 10.14(a), and 10.15(a). The table compares five generation configurations, non-join-shared query networks (NonJoinS), match-plan without canonicalization (MPLan_NCanon), sharing-selection without canonicalization (NonCanon), match-plan with canonicalization (MatchPlan), and sharing-selection with canonicalization (AllSharing). AllSharing is the best.

# of queries	NonJoinS	NonCanon	MatchPlan	AllSharing
100	1650	188	116	116
200	3434	574	332	332
300	5914	2435	688	637
400	8524	3955	1122	1003
500	11759	4356	1764	1408
600	15200	4764	2412	1813
700	19152	4957	2797	2007
768	21836	5081	3057	2138

Table C.7: Weighted query network sizes on FED query networks, shown in Figure 10.16(a). The table compares four generation configurations, non-join-shared query networks (NonJoinS), sharing-selection without canonicalization (NonCanon), match-plan with canonicalization (MatchPlan), and sharing-selection with canonicalization (AllSharing). AllSharing is the best.

# of queries	NonJoinS	Mplan_NCanon	NonCanon	MatchPlan	AllSharing
100	18.4765	12.195	12.2035	12.047	12.3355
200	42.4295	30.75	30.9765	30.1405	30.5235
300	67.75	52.656	49.094	48.32	35.5855
400	89.5235	61.4225	58.945	55.828	53.727
500	107.2655	67.688	66.281	64.4925	60.6
565	114.8045	69.765	66.7975	67.8515	61.641

Table C.8: Execution times on MED query networks, shown in Figures 10.13(e), 10.14(e), and 10.15(e). The table compares five generation configurations, non-join-shared query networks (NonJoinS), match-plan without canonicalization (MPLan_NCanon), sharing-selection without canonicalization (NonCanon), match-plan with canonicalization (MatchPlan), and sharing-selection with canonicalization (AllSharing). AllSharing is the best.

# of queries	NonJoinS	NonCanon	MatchPlan	AllSharing
100	2414	1485	1401	1401
200	4760	2807	2591	2591
300	7245	4011	3787	3750
400	9799	5193	4936	4875
500	12403	6295	6064	5961
565	14083	7025	6786	6673

Table C.9: Weighted query network sizes on MED query networks, shown in Figure 10.16(e). The table compares four generation configurations, non-join-shared query networks (NonJoinS), sharing-selection without canonicalization (NonCanon), match-plan with canonicalization (MatchPlan), and sharing-selection with canonicalization (AllSharing). AllSharing is the best.