Predicate Indexing for Incremental Multi-Query Optimization

Chun Jin and Jaime Carbonell

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

Abstract. We present a relational schema that stores the computations of a shared query evaluation plan, and tools that search the common computations between new queries and the schema, which are the two essential parts of the Incremental Multiple Query Optimization (IMQO) framework we proposed to allow the efficient construction of the optimal evaluation plan for multiple continuous queries.

1 Introduction

Multi-query optimization (MQO) [15], namely, finding the shared optimal query evaluation plan for multiple queries, has been widely studied, because sharing intermediate results among multiple queries can lead to significant computation reduction. But as NP-hard as it is, MQO usually has to employ heuristics to trade off between the optimality and the optimization time. MQO is particularly essential to stream databases since they tend to run multiple long-lived continuous queries concurrently, and the cost of finding the optimal plan will be amortized by continuous query evaluation. However, in reality, stream DB queries usually arrive at different times, as opposed to the assumption of synchronous arrival based on which the traditional MQO conducts the optimization as a one-shot operation. To cope with the query asynchrony and mitigate the NP-hardness, we propose a new approach, Incremental Multi-Query Optimization, by adding new query computation *incrementally* to the existing query plan with heuristic local optimization.

IMQO takes 4 steps to generate the new query evaluation plan R^*, given the existing plan R and a new query Q: 1. index R's computations and store them in persistent data structures $\mathcal{R}$; 2. identify common computations $\mathcal{C}$ between Q and R by searching $\mathcal{R}$; 3. select the optimal sharing path $\mathcal{P}$ in R that computes $\mathcal{C}$; and 4. expand R to R^* with the new computations $Q - \mathcal{C}$ that compute the final results for Q.

In this paper, we focus on the *Index* and *Search* (Steps 1 & 2) for selection-join-projection (SJP) queries; this presents the most thoroughly investigated effort so far on the most common query types (SJP). Previous work [11,10,9] discussed other query types, the sharing strategies (Step 3), and the actual continuous query plan construction (Step 4). The constructed plan will match the stream data on the fly to produce continuous query results.

A. An et al. (Eds.): ISMIS 2008, LNAI 4994, pp. 339–350, 2008.

Algorithm 1. IMQO
Input: R, Q
Output: R^*
Procedure:
1. $\mathcal{R} \Leftarrow Index(R)$);
2. $\mathcal{C} \Leftarrow Search(Q, \mathcal{R})$;
3. $\mathcal{P} \Leftarrow SelectSharing(\mathcal{R}, \mathcal{C})$;
4. $R^* \Leftarrow Expand(R, \mathcal{P}, Q - \mathcal{C})$.

Fig. 1. IMQO Algorithm

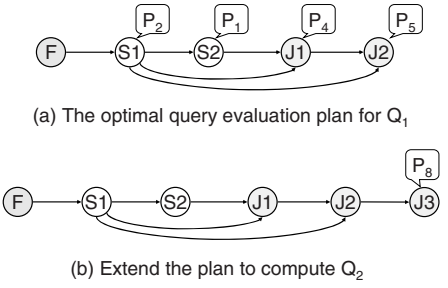

Fig. 2. Query Evaluation Plans

In our approach, the index and search are conducted on the relational schema that stores the query plan information and is updated incrementally when new computation is added into the plan. To design the schema, we need to consider: what types of plan information should be indexed; and how to index them to permit efficient search?

For the first question, the answer depends on the plan structure. We adopt the one widely used in traditional query optimization [14]; in particular, a plan is a directed acyclic graph, constructed from the where-clause which is in conjunctive normal form (CNF). As the results, the schema indexes literal predicates, disjunctions of literals (*OR predicate*, or *ORPred*), the conjunctions of the disjunctions (*Predicate Set*, or *PredSet*), and the plan topology. The tools search the equivalence and subsumption relationships at the literal, ORPred, and PredSet layers, and the sharable nodes in the plan topology. This covers the common SJP queries.

For the second question, our solution is modeling the computations using the ER (Entity Relationship) approach and then transforming the ER model to the relational model. This allows us to utilize the ER modeling power to analyze the query plan structure, and separate the scalability solution from information access logic. In particular, the indexing and searching algorithms are realized with database updates and queries to the relational system catalogs; and the efficiency of the catalog access is gained from intrinsic database functionalities, such as primary key indexing on catalog tables.

We integrated the indexing schema and tools into ARGUS [9], a stream processing system that was built upon the commercial DBMS Oracle to process multiple complex continuous queries, and is under planning for insertion into government agency systems RDEC (*www.globalsecurity.org/military/agency/army/cecom-rdec.htm*). Empirical evaluation on ARGUS shows that with moderate acceptable cost, the schema leads to constructing shared plans that achieve up to hundreds of times speed-up in query evaluation. Due to space limit, see [9].

The proposed schema is general, being usable in stream DB systems, such as STREAM [12], Aurora [1], and NiagaraCQ [5], with minor code change. Part of the schema, in particular, the canonicalization, and the indexing and searching at the literal and ORPred layers, can also be used to enhance the common computation identification on flow-based stream processing architectures, such as TelegraphCQ.

In the remaining of the paper, we discuss the related work in Section 2, present two query examples in Section 3 to illustrate the types of computations to be indexed, present the schema design in Section 4, and conclude in Section 5.

2 Related Work

In this section, we discuss the related work on query computation indexing and searching that has been done in MQO [15], view-based query optimization (VQO)[13,2], and stream databases [12]. The major difference is that our work employs the systematic approach to analyze and model the computation and extensively expands the scope of the previous work.

Two common approaches to query indexing in MQO and VQO are bottom-up query graph search [6,3,16]. and top-down rule-based filtering[8]. The first approach performs the common predicate search with one-by-one string match through one query graph after another. The second approach identifies the sharable views by filtering out irrelevant ones with different fine-tuned tests, such as excluding the ones not containing all the required columns and the ones with more restrict range predicates.

IMQO is different from MQO and VQO. MQO focuses on one-shot optimization where queries are assumed to be available all at a time and usually uses query graph. VQO identifies the optimal materialized views to speed up the query evaluation and uses both approaches mentioned above. Therefore, MQO and VQO do not index plan structure. But IMQO indexes all materialized intermediate results across the entire shared query plan, which allows full sharability search in an efficient way.

All the known stream database systems endorse computation sharing in certain ways. But so far, most focus on the sharing strategy (Step 3), plan expansion (Step 4), and engine support. NiagaraCQ [5] focused on the strategies of selecting optimal sharing paths; Aurora [1] supported shared-plan construction with heuristic local optimization with a procedural query language; TelegraphCQ [4] realized the sharing and computation expansion on a flow-based stream processing architecture; and STREAM [12] implemented the stream processing architecture that supports shared plans.

To our knowledge, only NiagaraCQ and TelegraphCQ realized the computation indexing and searching to support IMQO on declarative queries. They applied a simple approach that identifies identical predicates and subsumptions on selection predicates which must be in the form of $Attribute\ op\ Constant$.[1] In contrast, our work supports full range of query predicates and allows equivalent ones in different format to be identified.

3 Query Examples

In this section, we present two query examples to illustrate the types of computations. The queries are formulated on the FedWire database (FED). FED contains one single data stream, comprised of FedWire money transfer transaction records. A transaction, identified by *tranid*, contains the transaction type *type_code*, date *tran_date*, amount *amount*, originating bank *sbank_aba* and account *orig_account*, and receiving bank *rbank_aba* and account *benef_account*. Consider a query Q_1 on big money transfers for financial fraud detections.

[1] Subsumption is a containment relationship between predicates. The predicate p_1 subsumes the predicate p_2, if p_2 implies p_1, denoted as $p_2 \rightarrow p_1$; then p_2 can be evaluated from the results of p_1, which reduces the amount of data to be processed.

Example 1. The query links big suspicious money transactions of type 1000 or 2000, and generates an alarm whenever the receiver of a large transaction (over $1,000,000) transfers at least half of the money further using an intermediate bank within 20 days. The query can be formulated as a 3-way self-join on F, the transaction stream table:

```
SELECT r1.tranid, r2.tranid, r3.tranid
FROM   F r1, F r2, F r3
WHERE  (r1.type_code = 1000 OR
        r1.type_code = 2000)                    –p1
AND    r1.amount > 1000000                      –p2
AND    (r2.type_code = 1000 OR
        r2.type_code = 2000)                    –p3
AND    r2.amount > 500000                       –p4
AND    (r3.type_code = 1000 OR
        r3.type_code = 2000)                    –p5
AND    r3.amount > 500000                       –p6
AND    r1.rbank_aba = r2.sbank_aba              –p7
AND    r1.benef_account = r2.orig_account       –p8
AND    r2.amount > 0.5 * r1.amount              –p9
AND    r1.tran_date <= r2.tran_date             –p10
AND    r2.tran_date <= r1.tran_date + 20        –p11
AND    r2.rbank_aba = r3.sbank_aba              –p12
AND    r2.benef_account = r3.orig_account       –p13
AND    r2.amount = r3.amount                    –p14
AND    r2.tran_date <= r3.tran_date             –p15
AND    r3.tran_date <= r2.tran_date + 20;       –p16
```

We added two predicates $p4$ and $p6$ into the query. They can be inferred automatically [11] from $p2$, $p9$, and $p14$, and their data filtering improves the performance.

A continuous query evaluation plan should materialize some intermediate results on historical data, so they can be used to compute new results without repetitive computations over them (Rete-based query evaluation [11,5]). The materialized results can also be used for sharing among multiple queries. An effective materialization strategy is pushing down highly-selective selection predicates and materializing their results, so joins can be efficiently evaluated from much less intermediate results upon new data arrivals [7,5,11].

On the other hand, materialization should be used with caution because of the entailed disk access cost. An effective heuristic to avoid unnecessary materialization is grouping predicates based on the tables they reference and materializing the predicate groups (PredSet) [11], instead of each single predicate. So we get the PredSets: $P_1 = \{p_1, p_2\}$, $P_2 = \{p_3, p_4\}$, $P_3 = \{p_5, p_6\}$, $P_4 = \{p_7, p_8, p_9, p_{10}, p_{11}\}$, and $P_5 = \{p_{12}, p_{13}, p_{14}, p_{15}, p_{16}\}$.

Figure 2(a) shows the optimal plan to evaluate the query. We assume the selection PredSets, P_1, P_2, and P_3, are highly-selective, thus they are pushed down in the plan. Since PredSets P_2 and P_3 are equivalent, they share the same node $S1$. P_1 is subsumed by P_2 or P_3, thus P_1 can be evaluated from $S1$, instead of being evaluated from the source node F, shown as node $S2$. The subsumption sharing is useful since it reduces the amount of data to be processed to obtain $S2$. The results of P_4 and P_5 are also materialized to facilitate the efficient joins. If we assume that P_4 and P_5 are equally selective, then P_4 is evaluated first, since the size of the input to P_4 is less than that of P_5.

Consider the second query Q_2. Q_2 is similar to Q_1 except that the time span is 10 days instead of 20 days. Thus predicates p_{11} and p_{16} are substituted by $p_{17} = \{r2.tran\_date <= r1.tran\_date + 10\}$ and $p_{18} = \{r3.tran\_date <= r2.tran\_date + 10\}$, respectively; and PredSets P_4 and P_5 are by $P_6 = \{p_7, p_8, p_9, p_{10}, p_{17}\}$ and $P_7 = \{p_{12}, p_{13}, p_{14}, p_{15}, p_{18}\}$.

Since P_6 and P_7 are subsumed by P_4 and P_5 respectively, the final results of Q_2 can be evaluated from $J2$ with a selection PredSet $P_8 = \{p_{17}, p_{18}\}$, shown in Figure 2(b) as node $J3$.

From the examples, we see that the indexing schema should store several types of plan information, including literals, ORPreds, PredSets, and the plan topologies, and the searching tools should recognize the equivalence and subsumptions at the three predicate layers, and their associations with the plan topologies.

4 Predicate Indexing Schema

We describe the computation indexing schema and searching algorithms in this section. Firstly, we model the computations using the ER model methodology; then we present the relevant problems and their solutions; and finally, we derive the relational model from the first two steps.

4.1 ER Model for Plan Computations

We model the computations of a query evaluation plan as a 4-layer hierarchy. From top to bottom, the layers are topology layer, PredSet layer, ORPred layer, and literal layer.

Figure 3 shows the hierarchy for the two nodes $S1$ and $S2$ in Figure 2. For the equivalent PredSets P_2 and P_3, only P_2 is shown. For the equivalent ORPreds p_1 and p_3, only p_3 is shown, while p_1 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 and sharability between them.

The hierarchy can be presented in an ER model, as shown in Figure 4. Before transforming the ER model to the relational model, we address several issues in Sections 4.2-4.6, including rich syntax (equivalent predicates expressed differently), self-join canonicalization, subsumption identification, and topology indexing. Then the solutions are implemented in the final relational model, see Section 4.7.

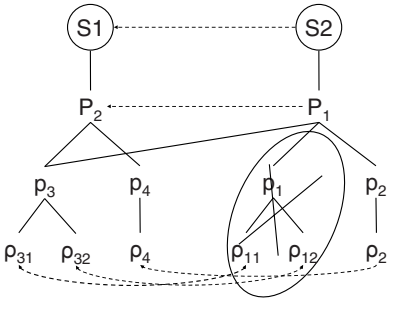

Fig. 3. Computation hierarchy. $\rho_{11} = \rho_{31}$: $type\_code = 1000$, $\rho_{12} = \rho_{32}$: $type\_code = 2000$, $\{\rho_2\} = p_2$, $\{\rho_4\} = p_4$.

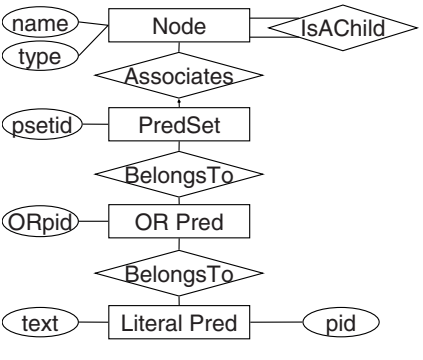

Fig. 4. Hierarchy ER model

4.2 Rich Syntax and Canonicalization

The first obstacle in the common computation identification is that semantically-equivalent literals can be expressed in different syntactic forms. For example, $t1.a < t2.b$ can also be expressed as $t2.b > t1.a$. A simple string match can not identify such equivalence. To solve the problem, 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.

But subsumption can still not be identified by the exact string match. For example, $t1.a > 5$ subsumes $t1.a \geq 10$. To address the problem, we apply a triple-string canonical form. For a literal ρ, we use $\gamma(\rho)$ to denote its operator, and use $L(\rho)$ and $R(\rho)$ to denote its left-hand-side and right-hand-side expressions, respectively. So ρ can be written as a triplet $L(\rho)\gamma(\rho)R(\rho)$. By making $L(\rho)$ a canonical expression of column references without constant terms, and $R(\rho)$ a canonical constant without any column references, such as $\rho : t1.a + 2 * t2.b > 5$, the subsumption can be identified by exact string match on $L(\rho)$, and comparisons on $\gamma(\rho)$ and $R(\rho)$. We apply recursive rules to transform expressions to predefined canonical format; please see [9] for the details.

The time complexity of canonicalization is up to quadratic to the length of the predicates because of sorting. But this is not considered 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.

4.3 Self-join

To allow exact-string match for finding equivalence and subsumption, table references in canonical forms should use true table name, not table alias. For example, $p4$ and $p6$ in Q_1 should be canonicalized as $F.amount > 500000$, so the equivalence can be identified. This is all right for a selection predicate or join predicate on different tables, but problematic for a self-join predicate.

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. In the search, we enumerate the STA assignments in the canonical form to find the predicate match.

Self-joins also present problems in the PredSet and ORPred layers. For example, an ORPred p contains 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, we must identify its STA with respect to the self-join predicate ρ_2. Therefore, ρ_1's STA, $T1$ or $T2$, must be recorded when indexing p. Similar situation exists in PredSets where some ORPreds are single-table selections while others are self-joins. 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. In particular, a PredSet chooses one STA assignment, and propagates it down to the ORPred layer and then the literal layer.

A 2-way self-join condition, being a literal, ORPred, or PredSet, has two possible STA assignments. And a k-way self-join has $k!$ assignments. This means that a search algorithm may try up to $k!$ times to find a match. The factorial issue is intrinsic to self-join matching, but may be addressed heuristically. In our implementation, supporting 2-way joins, the search on a self-join PredSet stops when it identifies an equivalent one from the system catalogs. If both assignments lead to the identification of subsuming PredSets, the one that has less results (indicating a stronger condition) is chosen.

4.4 Subsumption on Literals

As discussed in previous sections, subsumption is important in computation sharing. It presents in all the three predicate layers. Given a new condition p, we want to identify all conditions in the existing plan that either subsume or are subsumed by p. The former directly leads to sharing, while the later can be used to re-optimize the plan.

In this subsection, we describe how subsumptions of comparison literal predicates are detected from the triple-string canonical forms. When $L(\rho_1) = L(\rho_2)$, the subsumption between the two literals, ρ_1 and ρ_2, may exist. It is determined by the relationships between $\gamma(\rho_1)$ and $\gamma(\rho_2)$, and between $R(\rho_1)$ and $R(\rho_2)$. 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-hand-side expressions.

Definition 1. *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 literals ρ_1 and ρ_2, such that $\rho_1 = L(\rho_1)\gamma_1 R(\rho_1)$, and $\rho_2 = L(\rho_2)\gamma_2 R(\rho_2)$, if $L(\rho_1) = L(\rho_2)$, and $O(R(\rho_1), R(\rho_2))$ is true, 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). Figure 5 shows the implemented subsumable triples. With this, look-up queries can be formulated to retrieve the indexed subsumption literals in constant time.

It can be shown that literal subsumptions identified this way have the following property.

Theorem 1. *If $\rho \rightarrow \rho_1$, $\rho \rightarrow \rho_2$, and the subsumptions are identifiable through the subsumable triples, then either $\rho_1 \rightarrow \rho_2$ or $\rho_2 \rightarrow \rho_1$ is true.*

4.5 Subsumption on ORPreds and PredSets

As discussed in previous sections, we want to identify subsumptions on ORPreds and then on PredSets. This is proceeded from the subsumptions identified at the literals; and the results identified on the PredSets are then used to find the sharable topologies.

Given the existing plan, R, we use R_{ORPred} and $R_{PredSet}$ to denote the set of all ORPreds and the set of all PredSets in R, respectively. Given a new query Q, we use P to denote a PredSet in Q, and use p to denote an ORPred in P, namely, $p \in P$, and $P \in Q$.

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

Fig. 5. Subsumable Triples (γ_1, γ_2, O). E is equal, D is decreasing, and I is increasing.

Algorithm 2. Subsumed_ORPreds
Input: p, R
Output: $SubsumedSet(p)$
Procedure: for each literal $\rho_i \in p$, $1 \leq i \leq l$
$S_{\rho_i} \Leftarrow \{p_{ijk} \Rightarrow \{\rho_{ij}\} \mid \rho_i \rightarrow \rho_{ij},\ \rho_{ij} \in p_{ijk},$
$p_{ijk} \in R_{ORPred},\ 1 \leq j \leq s,\ 1 \leq k \leq m\}$;
$I \Leftarrow \cap_{i=1}^{l} S_{\rho_i}$;
$SubsumedSet(p) \Leftarrow \{\}$;
for each key $p' \in keys(I)$
if $|elements(I, p')| == |p|$
$SubsumedSet(p) + \Leftarrow p'$;

Fig. 6. Subsumption Algorithm

The problem of identifying subsumptions on ORPreds is as follows. Given an ORPred p, such that $p \in P$, and $P \in Q$, we want to find all ORPreds $p' \in R_{ORPred}$, such that p is subsumed by, subsumes, or is equivalent to p'. Similarly, given the PredSet $P \in Q$, we find all PredSets $P' \in R_{PredSet}$, such that P is subsumed by, subsumes, or is equivalent to P'.

In the rest of this subsection, we focus on the algorithm, *Subsumed_ORPreds*, which finds all the ORPreds that subsume p. We use the algorithm as the example to describe the computation representation, data structure and its operation, and the algorithm logic. We also cover other subsumption identification algorithms, which can be realized by small modifications to *Subsumed_ORPreds*. Finally, we discuss the algorithm time complexity.

Representation. We assume that each ORPred p has l literals, $\{\rho_1, \cdots, \rho_l\}$, each literal ρ_i is subsumed by s indexed literals, $\{\rho_{i1}, \cdots, \rho_{is}\}$, and each indexed literal ρ_{ij} appears in m non-equivalent ORPreds, $\{p_{ij1}, \cdots, p_{ijm}\}$, as shown in the left-hand-side of Figure 7. l is related to typical types of queries registered into the system, and thus can be viewed as a constant parameter. Similarly, we assume that each PredSet has k ORPreds, each ORPred is subsumed by t indexed ORPreds, and each ORPred appears in n different PredSets.

The algorithms assume non-redundant representations on ORPreds and PredSets. In particular, for the ORPred case, the assumption says that given an ORPred p, either indexed in $\mathcal{R}$ or in the new query Q, any literal $\rho \in p$ does not subsume any other literal $\rho' \in p$. For example, if $p = \{\rho_1\ OR\ \rho_2\}$ is non-redundant, then neither $\rho_1 \rightarrow \rho_2$ nor $\rho_2 \rightarrow \rho_1$ holds. Non-redundant PredSet representation is defined similarly. The assumption assures that all the subsumptions can be found with a single pass of the $l * s * m$ ORPreds or $k * t * n$ PredSets, based on the Theorem 2.

Theorem 2. *If the non-redundant assumption holds, then the $s*m$ ORPreds, $\{p_{ijh} | 1 \leq j \leq s, 1 \leq h \leq m\}$, whose literals subsume ρ_i, are different to each other.*

Proof. By contradiction, assume there are j_1, j_2, h_1, and h_2, such that $p_{ij_1h_1} \equiv p_{ij_2h_2}$. Then $\rho_{ij_1} \in p_{ij_1h_1}$, and $\rho_{ij_2} \in p_{ij_1h_1}$. According to Theorem 1, either $\rho_{ij_1} \rightarrow \rho_{ij_2}$ or $\rho_{ij_2} \rightarrow \rho_{ij_1}$ holds, which contradicts to the non-redundant assumption.

Note that the ORPreds across different literal predicates, such as $p_{i_1j_1k_1}$ and $p_{i_2j_2k_2}$, where $i_1 \neq i_2$, could be legitimately equivalent, and potentially are the targeted results the algorithms look for. The similar property can also be proven on the PredSet representations.

Corollary 1. *If the non-redundant assumption holds, then $s * m \leq |R_{ORPred}|$, where $|R_{ORPred}|$ is the number of ORPreds in R.*

The corollary follows from the theorem immediately. Generally, $s * m \ll |R_{ORPred}|$, since on average, ρ_i and its subsumed literals $\{\rho_{i1}, \cdots, \rho_{is}\}$ present a narrow set of semantics and only appear in a small portion of R_{ORPred}.

Data Structure. The algorithms use a data structure called 2-level hash set (2HSet) built up for literals or ORPreds to record the subsumption relationships. A 2HSet S is a set of sets, containing a set of hash keys, denoted as $keys(S)$, and each hash key $p \in keys(S)$ pointing to a set of elements, denoted as $elements(S, p)$. $keys(S)$ and all $elements(S, p)$ are hashed for constant-time accesses. For *Subsumed_ORPreds*, a 2HSet S_{ρ_i} records the ORPred set $\{p_{ijh} | 1 \leq j \leq s, 1 \leq h \leq m\}$, where each ORPred p_{ijh} in the set contains a literal ρ_{ij} that subsumes $\rho_i \in p$, as shown in Figure 7, where $keys(S_{\rho_i}) = \{p_{ijh} | 1 \leq j \leq s, 1 \leq h \leq m\}$, and $elements(S_{\rho_i}, p_{ijh}) = \{\rho_{ij}\}$.

We define a binary operation Υ-intersection $\cap_\Upsilon$ on 2HSets S_1 and S_2 to identify the intersection of their hash key sets, which represents the common part between the predicate conditions.

Definition 2. *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 $\forall p \in keys(S)$, $elements(S, p) = elements(S_1, p) \cup elements(S_2, p)$.*

Υ-intersection preserves only the hash keys that appear in both S_1 and S_2. For any preserved hash key p, its $elements$ set is the union of p's $elements$ sets in S_1 and S_2. Υ-intersection can be computed in the time of $O(|keys(S)| * L)$ where S is the probing operand, either S_1 or S_2, in the implementation, and L is the average number of elements in $elements(S, p)$ for all $p \in keys(S)$. In Figure 7, the time of Υ-intersecting two 2HSets is $O(s * m)$.

Subsumption Algorithms. *Subsumed_ORPreds*, as shown in Figure 6, finds all ORPreds in R_{ORPred} that subsume p. It constructs all S_{ρ_i}, $1 \leq i \leq l$, Υ-intersects them to generate the final 2-level hash set I, and checks which remaining ORPreds in I subsume p. $|elements(I, p')|$ is the number of elements in $elements(I, p')$. And $|p|$ is the number of literals in p. The check condition $|elements(I, p')| = |p|$ means that if each literal in p is subsumed by some literal in p', then p is subsumed by p'.

A similar algorithm, $Subsume\_ORPreds$, finds all ORPreds in R_{ORPred} that p subsumes. The two differences from $Subsumed\_ORPreds$ are that the 2HSets are

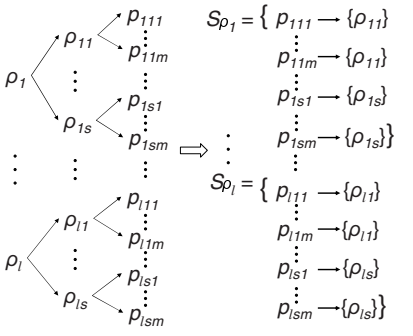

Fig. 7. ρ_i is subsumed by ρ_{ij}, $\rho_{ij} \in p_{ijk}$, $1 \leq i \leq l, 1 \leq j \leq s, 1 \leq k \leq m$. The information can be stored in 2-level hash sets.

PredIndex	PSetIndex	SelectionTopology
ORPredID	PredSetID	Node
LPredID	ORPredID	DirectParent
LeftExpr	STA	DPredSetID
Operator	**JoinTopology**	SVOA
RightExpr	Node	SVOAPredSetID
Node1	DirectParent1	JVOA1
Node2	DirectParent2	JVOA2
STA	DPredSetID	JVOAPredSetID
UseSTA	JVOA1	IsDISTINCT
	JVOA2	
	JVOAPredSetID	
	IsDISTINCT	

Fig. 8. System Catalogs

constructed from the literals that are subsumed by p's literals, and the final check condition is $|elements(I, p')| = |p'|$, meaning 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 at the PredSet layer. In that case, the hash keys are the PredSet IDs and the elements are ORPred IDs. The final check conditions dictate that a PredSet P is subsumed by another P' if P is subsumed by all ORPreds in P'. Identifying equivalence is easy given the identified subsuming and subsumed 2HSets; it is the unique ORPred or PredSet that is in the intersection of the two.

The algorithms guarantee that no redundant ORPreds or PredSets will be introduced into indexing as long as the non-redundancy assumption holds on queries.

Time Complexity. The time complexity of the ORPred-layer algorithms is $O(l*s*m)$, and that of the PredSet-layer is $O(k*l*s*m+k*t*n)$ which includes the k calls of the former. Note that $t*n \leq |R|$ given the non-redundancy assumption. $|R|$ is the number of the searchable PredSets in R and the number of nodes in R. Generally, $t*n \ll |R|$ since p usually appears only in 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.

4.6 Topology Connection

PredSets are associated with nodes. Assume that PredSet P is associated with node N, then P bears the topological connections between N and N's ancestor set $\{A\}$. In particular, the results of N are obtained by evaluating P on $\{A\}$.

The node N may be associated with multiple PredSets depending on the different types of ancestors. We define three types of ancestors for each node, direct parents ***(DParents)***, selection very original ancestor ***(SVOA)***, and join very original ancestors ***(JVOA)***.

Definition 3. *A node N's **DParents** are the set of nodes that have an edge pointing to N.*

Definition 4. *A selection node N's **SVOA** is N's closest ancestor that is either a join node or a base stream node. A join node or a base stream node N's **SVOA** is itself N.*

Definition 5. *A join node N's **JVOA**s are the closest ancestor nodes that are either join nodes (but not N) or base stream nodes. A selection node N's **JVOA**s are N's SVOA's JVOAs. And a base stream node's **JVOA** is NULL.*

We record all the three ancestor types and their associated PredSets. Each type plays an important role. DParents is necessary and sufficient to construct the plan execution code. SVOAs and JVOAs present local topological connections within and across one join depth, respectively. Their presence allows a single lookup per join depth, avoiding the chained search through DParents, to find the sharable computation.

4.7 Relational Model for Indexing

Now we convert the ER model to the relational model. A simplified version is shown in Figure 8. We made three adjustments. First, only 2-way joins are supported. Supporting multi-way joins requires a small amount of work to revise the indexing schema and the searching tools, but requires much more work in sharing strategies. In particular, multi-way joins bring back the NP-hardness, and requires more advanced heuristic optimization techniques. This will be a future work. Second, 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 third adjustment is splitting the node topology indexing relation (The Node entity in the ER model) to two, namely, *SelectionTopology*, and *JoinTopology*, based on the observation that the topology connections on selection nodes and on join nodes are quite different.

5 Conclusion and Future Work

As part of the IMQO framework, a comprehensive computation indexing schema and a set of searching tools were presented. The schema stores shared plan computations in relational system catalogs, and the tools search the common computations between new queries and the system catalogs. We implemented the schema and tools on ARGUS to support efficient processing of a large number of complex continuous queries. The empirical evaluation on ARGUS demonstrated up to hundreds of times speed-up for multiple query evaluation via the shared plan construction [9]. The techniques would also be very useful in other IMQO-supported stream databases.

For future work, immediate extensions are supporting multi-way joins, local restructuring upon new query arrivals, and adaptive local re-optimization upon dynamic con-

gestion detections. It is also interesting to support more advanced sharing strategies in the IMQO setting, such as identifying the minimum cover of disjoint ranges and utilizing constraints, such as foreign key constraints [8].

Acknowledgments

This work was supported in part by DTO/ARDA, NIMD program under contract NMA401-02-C-0033. The views and conclusions are those of the authors, not of the U.S. government or its agencies. We thank Christopher Olston, Phil Hayes, Jamie Callan, Minglong Shao, Santosh Ananthraman, Bob Frederking, Eugene Fink, Dwight Dietrich, Ganesh Mani, and Johny Mathew for helpful discussions.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB J. 12(2), 120–139 (2003)
2. Blakeley, J.A., Coburn, N., Larson, P-Å.: Updating derived relations: Detecting irrelevant and autonomously computable updates. ACM Trans. Database Syst. 14(3), 369–400 (1989)
3. Chakravarthy, U.S., Minker, J.: Multiple query processing in deductive databases using query graphs. In: VLDB, pp. 384–391 (1986)
4. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR (January 2003)
5. Chen, J., DeWitt, D.J., Naughton, J.F.: Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In: ICDE, pp. 345–356 (2002)
6. Finkelstein, S.J.: Common subexpression analysis in database applications. In: SIGMOD Conference, pp. 235–245 (1982)
7. Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence 19(1), 17–37 (1982)
8. Goldstein, J., Larson, P.-Å.: Optimizing queries using materialized views: A practical, scalable solution. In: SIGMOD Conference (2001)
9. Jin, C.: Optimizing Multiple Continuous Queries. Ph.D. Thesis CMU-LTI-06-009, Carnegie Mellon University (2006)
10. Jin, C., Carbonell, J.G.: Incremental aggregation on multiple continuous queries. In: ISMIS, pp. 167–177 (2006)
11. Jin, C., Carbonell, J.G., Hayes, P.J.: Argus: Rete + dbms = efficient persistent profile matching on large-volume data streams. In: ISMIS, pp. 142–151 (2005)
12. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: CIDR (2003)
13. Roussopoulos, N.: View indexing in relational databases. ACM Trans. Database Syst. 7(2), 258–290 (1982)
14. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD Conference, pp. 23–34 (1979)
15. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. 13(1), 23–52 (1988)
16. Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., Urata, M.: Answering complex sql queries using automatic summary tables. In: SIGMOD Conference, pp. 105–116 (2000)