Multiple Continuous Queries Evaluation over Data Streams
Hong Kyu Park and Won Suk Lee
Department of Computer Science, Yonsei University
134 Shinchon-dong Seodaemun-gu
Seoul, 120-749, Korea
+82-2-2123-2716

Abstract: Query processing for data streams should be continuous and rapid, which requires strict time constraint. In most previous researches, in order to guarantee this constraint, the evaluation order of join predicates in a continuous query is optimized by a greedy. However, the greedy strategy traces only the first promising plan, so that it often finds a sub-optimal plan. This paper proposes an improved scheme called an Adaptively Sharing-based Extended Greedy Algorithm (A-SEGO). Given continuous queries with multiple join operations, they simultaneously trace a set of promising plans to reduce the possibility of producing a sub-optimal plan. Also it can control the time to optimize continuous queries depending the current processing load by controlling the number of traced plans. Experiment results illustrate the performance of the A-SEGO in various stream environments.

Keywords: Data Streams, Multi-way join Query, Multiple Query Optimization, Query processing, Greedy

1 Introduction
A data stream is a massive unbounded sequence of data elements continuously generated at a rapid rate. It can be adapted in many applications such as web click monitoring, sensor data processing, network traffic analysis, etc. In these applications, continuous queries are generally used to monitor the ongoing information of data streams. They are registered in advance and should produce their result continuously and repeatedly whenever new data elements of data streams are generated. They should be evaluated as quickly as possible in order to process data streams in real-time. For this, query optimization is essential. However, traditional techniques used in DBMSs are not effective for data stream applications for the following reasons. First, the way to measure the cost of operators in queries over data streams is different from that in the database because processing continuous queries needs new techniques such as grouped filter[8], windowed join[15], MJoin[1], etc. and also is based in memory whereas traditional processing is based on a disk. Second, continuous queries should be continually re-optimized because of the unpredictable characteristics of data streams[1]. For these reasons, many previous researches[1,4,5,6,8,9] have focused on devising a method for optimizing the execution of continuous queries.

This paper addresses a query optimization scheme to produce the optimal execution plan for multiple queries which considers sharing the common predicates in the queries. Especially for multi-way join queries with multiple join predicates, finding the optimal execution plan is problematic and important since a join operation is more expensive than single-input operations such as selection and projection because it is a blocking operator even though it is processed as non-blocking operator with a window. Therefore, when continuous queries are concurrently active over common streams, sharing common join predicates is very attractive.

When we optimize multiple queries in data stream environments, we should consider the optimality of the optimized plan as well as the time to optimize since continuous queries should be processed in real-time. Consequently, previous approaches have employed a greedy strategy with respect to either the result size[4] or selectivity[5] of a join predicate. Even if the strategy can optimize continuous queries very quickly with almost no overhead, it is apt to produce a sub-optimal plan because there is no backtracking mechanism to undo a non-optimal intermediate order of join operations. However, employing a backtracking mechanism in data stream environment is impractical because it has a huge overhead. Our approach traces a set of promising plans simultaneously whereas a greedy strategy traces only one promising plan. It can reduce the possibility of producing a sub-optimal plan. In addition, the number of traced plans can be controlled depending on the current processing load. This means that we can flexibly control the time to optimize the set of continuous queries. The number of traced plans is determined by the cost statistics of multiple queries, which is continuously monitored and stored as average and standard deviation, and a user-defined variable $k$ (range variable). Our approach can capture the entire spectrum from a greedy optimization to full optimization depending the value of the variable.
2 Related Work

There are two common techniques for processing a join operator. The first technique, a pipelined processing tree (PPT) [17] uses a multi-way nested loop to generate the result of a multi-way join query without materializing any intermediate result. The second technique is a binary linear processing tree (BLPT) [17] that materializes intermediate results. A representative research of the first technique is a MJoin [1]. The operator is employed by many previous researches [4,5,8]. When many continuous queries are registered in DSMS, sharing same join conditions in several queries is helpful for decreasing a system load. However, MJoin operation is hard to share the operators because it processes all join conditions in a query at one time in the single MJoin operator. It produces the result of a multi-way join query without any tree-structured plan.

STREAM [5] uses the A-Greedy algorithm to adaptively determine the evaluation order of an MJoin operator to keep minimizing the processing cost of a multi-way join query [14,15]. In [18], A-Caching focuses specifically on the problem of adaptive placement and removal of caches in an MJoin operator to optimize join performance. It provides algorithms for selecting caches, monitoring their costs and benefits in current conditions, allocating memory to caches, and adapting as conditions change. Eddy [11] accomplishes very fine-grained adaptivity without employing any fixed execution plan. Its query execution model routes input and intermediate result tuples among query operators by making independent routing decisions for each tuple to reflect the dynamically changing runtime selectivity of each query condition. In [7], as a modified architecture of the original Eddy that uses SteMs, a query operator called a STAIR is introduced to perform a join operation. STAIRs employs the second technique, BLPT. Even though it solves the drawbacks of the original eddy, they address the problems caused by allowing query state to be modified and migrated across STAIRs during query execution. STAIRs also solved the problem that the ability of an eddy to adapt is constrained by the state that gets accumulated in the query operators.

3 Preliminaries

3.1 The Join Cost Model

In order to define the cost of a binary join operation, we use a unit-time based cost model [12]. In the execution of a binary linear processing tree, the result of one binary join operation should be materialized because it can be an operand for the next binary join operation. Consequently, the result size of a binary join operation affects the overall evaluation cost of a multi-way join query, so that the unit-time based cost model is modified for a multi-way join query in this paper. The terms used in the modified cost model are described in Table 1. A join operation R ≻◁ S should be executed when a new tuple arrives from either of its two operand streams R and S. Accordingly, the evaluation cost $C(R ≻◁ S)$ of a binary join operation R ≻◁ S for newly arriving tuples in a unit time should be symmetrically defined as follows.

$$C(R ≻◁ S) = (\lambda_k + \lambda_s) \times (\frac{W_R \times \lambda_k}{|S|} + \frac{W_S \times \lambda_s}{|R|})$$ (1)

The generating cost of the result of a binary join operation, $C(R ≻◁ S)$ is proportional to the size of the result and is defined as follows.

$$C(R ≻◁ S) = \lambda_k \lambda_s (W_R + W_S)$$ (2)

The total evaluation cost of a binary join operation on data streams R and S is calculated by adding the evaluation cost to the generating cost. If the operation is shared by several queries, the cost should be divided by the number of queries with the operation. Therefore the total evaluation cost of the join operation is defined as follows:

$$C(R ≻◁ S) = C_i(R ≻◁ S) + C_i(R ≻◁ S)/Task(R ≻◁ S)$$ (3)

| $\lambda_k$ | Input rate of stream R |
| $W_R$ | Window size of stream R |
| $|R|$ | Hash table size of stream R |
| $\lambda_s$ | New tuples inserted into stream R |
| $\sigma_{R,S}$ | Join selectivity of $R ≻◁ S$ |
| $C_i(R ≻◁ S)$ | Join evaluation cost per unit time |
| $C_i(R ≻◁ S)$ | Result processing cost per unit time |
| $C(R ≻◁ S)$ | Total cost of $R ≻◁ S$ |
| Task($R ≻◁ S$) | The # of queries with the predicate $R ≻◁ S$ |

3.1 Monitoring Join Properties

A multi-way join query can be evaluated by a number of different execution plans. In order to exactly optimize continuous queries with a cost-based optimizer, the properties of all the join operations (join selectivity, arrival rates of input streams, etc.) in all the plans should be monitored. They are maintained in the catalog, Cost_Statistics. The outstanding feature of Cost_Statistics is that the costs of all the join operations in the table don’t represent the cost at only one moment. The costs are calculated from the properties monitored so far and stored as the average cost and its standard deviation. This means that it can represent how the costs have fluctuated. If the cost of a join operation fluctuated widely in the past, the standard deviation becomes large, cost_statistics formally maintains properties of join operations such as $J_q^m$, $\mu(C(J_q^m))$, $\sigma(C(J_q^m))$, count($J_q^m$) and $qBit(J_q^m)$. $J_q^m$ denotes a join predicate, $\mu(C(J_q^m))$ and
\( \sigma(C(J_q^m)) \) respectively denote the average cost and standard deviation of the \( J_q^m \). \( \text{count}(I_q^m) \) denotes how many times the join operation \( J_q^m \) has been executed before and \( q \text{Bit}(I_q^m) \) denotes the queries which have \( J_q^m \).

**Definition 1.** (Sharable Join Operation (SJO)) For \( n \) multi-way join queries \( \text{QSet} = \{Q_1, Q_2, \ldots, Q_n\} \), each query has multiple join operations \( Q_k = \{J_1^k, \ldots, J_m^k\} \). \( J_m^k \) is a sharable join operation if \( J_m^k \) has been executed \( q \text{Bit}(I_q^m) \) and \( Q_n \) is a sharable query set (SQS). □

4 Adaptively Sharing-based Extended Greedy Optimization (A-SEGO)

This method traces a set of promising plans simultaneously while a greedy strategy traces only one promising plan. It can not only reduce the possibility of producing a sub-optimal plan, but can also control the number of traced plans depending on the current processing load. To describe a part of an execution plan, the term \( m\text{-subplan} \) is defined formally in Definition 2.

**Definition 2.** (\( m\text{-subplan} \)) For multi-way join queries \( \text{QSet} = \{Q_1, Q_2, \ldots, Q_n\} \), each query has multiple join operations \( Q_k = \{J_1^k, \ldots, J_m^k\} \). An \( m\text{-subplan} \) of the queries is a partial execution plan with \( m \) join operations \( sp^m = \{J_m^1, \ldots, J_m^n\} \
4 Experiments
This section provides experimental results of the proposed algorithm. All the algorithms were implemented in C, and all the experiments were executed on a Pentium 4 CPU 2.66GHz system with 1G RAM. The system runs Linux with the 2.4.5 kernel and gcc 3.3.2. In order to analyze the performance of the A-SEGO, multiple queries(QSet) and synthetic datasets are generated. The QSet contained join queries over the N sources and had equi-join predicates between each pair of sources. We generate QSets_ratios = \{Q 1,…Q n\}, where ratio_i is the ratio of SJO and is defined as:

\[
\sum_{i=1}^{m} \text{The # of Join Operations in Q}_i
\]

The default value of ratio_i is 0.2. The synthetic datasets simulated the Cost_Statistics of QSets_ratio. The input rate and the domain size of an operand data stream were uniformly selected within a pre-defined range, [I … d_max]. A window size and hash table size were applied to all data sources. All experiments were repeated 1000 times and then the average of measured values was calculated.

In Figure 2, 20 multi-way join queries with 6 join operations were performed for the 3 synthetic datasets in Table 3(Each dataset has different characteristics as shown in Table 3). Given a value of k, let Cost_k denote the cost of the execution plan generated by A-SEGO and Cost_opt denote the cost of the optimal plan generated. To observe the performance of the A-SEGO with respect to the value of a range variable k, we define the relative effectiveness E(k) of the A-SEGO as follows:

\[
E(k) = \frac{\text{Cost}_k - \text{Cost}_{opt}}{\text{Cost}_{opt}}
\]

The smaller the value of E(k) is, the closer to the optimal plan the execution plan generated by A-SEGO. Figure 2 shows that the value of E(k) decreased as the value of k increased and also the optimality of A-SEGO was degraded when the size of intermediate results increased with the level of join operation.

Table 2. DataSets

<table>
<thead>
<tr>
<th>DataSet1</th>
<th>10000</th>
<th>As the level of join operation deepens, the size of intermediate results decreases.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSet2</td>
<td>1000</td>
<td>The size of intermediate results is almost unchanged irrespective of the level of join operation.</td>
</tr>
<tr>
<td>DataSet3</td>
<td>100</td>
<td>As the level of join operation deepens, the size of intermediate results increases.</td>
</tr>
</tbody>
</table>

Fig. 2 The influence of the value of range variable

Fig. 3 Time – optimality trade-off

4 Conclusion
Most continuous queries are registrated in advance and should be processed continuously, so multiple query optimization is very important in data stream environments. In this paper we propose, in order to optimize multi-way join queries in a data stream...
environment, the **Adaptively sharing-based Extended Greedy Optimization (A-SEGO)**. The proposed method uses a *Cost_Statistics* table, which reflects the monitored costs of join operations as well as whether or not the join operations are shared by the queries, and traces a set of promising subplans simultaneously to reduce the possibility of producing a sub-optimal plan. We show how to control the number of simultaneously traced subplans flexibly with a range variable \( k \). As a result, it is possible to adaptively control the number of the subplans by changing the value of \( k \) depending on the current processing load. Experimental results show how efficiently **A-SEGO** can optimize multi-way join queries in various stream environments. **A-SEGO** can provide a nice trade-off between the optimality of a generated execution plan and its optimization time.

**ACKNOWLEDGEMENT**

“This work was supported by the Korea Science and Engineering Foundation (KOSEF) NRL Program grant funded by the Korea government(MEST) (No. R0A-2006-000-10225-0)”

References:


