PAPER

Adaptive Continuous Query Reoptimization over Data Streams

Hong Kyu PARK^{†a)}, Member and Won Suk LEE^{†b)}, Nonmember

SUMMARY A data stream is a series of massive unbounded tuples continuously generated at a rapid rate. Continuous queries for data streams should be processed continuously, so that a strict time constraint is required. In most previous research studies, in order to guarantee this constraint, the evaluation order of join predicates in a continuous query is optimized using a greedy strategy. However, because a greedy strategy traces only the first promising plan, it often finds a suboptimal plan. To reduce the possibility of producing a suboptimal plan, in this paper, we propose an improved scheme, *k-Extended Greedy Algorithm (k-EGA)*, that simultaneously examines a set of promising plans and reoptimize an execution plan adaptively. The number of promising plans is flexibly controlled by a user-defined range variable. The scheme verifies the performance of the current plan periodically. If the plan is no longer efficient, a newly optimized plan is generated. The performance of the proposed scheme is verified through various experiments to identify its various characteristics.

key words: data stream processing, multiway join query, multiple query optimization, greedy strategy, query processing

1. Introduction

Many recent research studies of emerging applications often need to deal with a data stream, which is a massive unbounded sequence of data elements continuously generated at a rapid rate. Such applications can be web click monitoring, sensor data processing, network traffic analysis, telephone recording and multimedia data processing. In these applications, a continuous query is generally used to monitor the ongoing information on data streams. It is registered in advance and produces results continuously and repeatedly whenever new data elements of data streams are generated. Since new data elements can be rapidly generated and should be processed in real time, a continuous query should be evaluated as quickly as possible. For this, query optimization is essential. However, query optimization techniques used in a traditional database management system are not effective for data stream applications because of the unpredictable characteristics of data streams [1]. For this reason, many previous research studies [1], [4]–[6], [8], [9] are focused on devising a method of optimizing the execution of continuous queries. A multiway join query contains multiple join operations.

Finding the optimal execution sequence of join opera-

tions in such a query is problematic since a join operation is very expensive. The execution plan of a multiway join query should be continuously reoptimized in data stream environment. For this purpose, properties such as the input rates of its operand data streams and the join selectivities of its join operations should be continuously monitored. However, such monitoring requires a huge runtime overhead. In [7], given a join operation, the domain sizes of its two join attributes are monitored to estimate its join selectivity. In [5], STREAM actually obtains the join selectivity of every join operation in every possible left-deep tree plan by additionally executing them for sampled tuples. Therefore, it can obtain more accurate statistics, but it must evaluate a number of plans in run time.

To estimate the current evaluation cost of a join operation, the past properties of the operation are used. Even though the properties of a join operation may be changed in runtime dynamically, they may not vary randomly. Instead, they tend to have valid ranges respectively. This means that the past properties of a join operation can be used to predict the current cost of the operation more accurately. For this purpose, the properties of join operations that are executed at least once are maintained in a table called *cost-Statistics*.

In order to continuously reoptimize the query, the time to generate the optimized plan of a multiway join query is also an important factor. Consequently, most approaches employ a greedy optimization strategy with respect to either the size of the result [4] or join selectivity [5]. Even though the greedy strategy can generate its solution very quickly, it is apt to produce a suboptimal plan. This is because there is no backtracking mechanism to undo the intermediate order of join operations when the order turns out to be nonoptimal. However, employing a backtracking mechanism in data stream environments is also inadequate because of the time constraint as mentioned above.

In this paper, we propose a query optimization scheme called *k-Extended Greedy Algorithm (k-EGA)* to produce the optimal execution plan of a multiway join query. As mentioned above, the greedy strategy traces only one promising subplan, whereas k-EGA traces a set of promising subplans simultaneously. Intuitively the cardinality of the set affects the optimality of the algorithm as well as optimization time. As it increases, the possibility of generating a more optimal plan is increased, but the time to produce the plan is increased as well. To identify the effective set of promising subplans, given a user-defined range variable *k*, the cost of each subplan is represented as a range based on the aver-

Manuscript received August 19, 2008.

Manuscript revised January 20, 2009.

[†]The authors are with the Department of Computer Science, Yonsei University 134 Shinchon-dong Seodaemun-gu Seoul, 120–749, Korea.

a) E-mail: gladiator11@database.yonsei.ac.kr

b) E-mail: leewo@database.yonsei.ac.kr

DOI: 10.1587/transinf.E92.D.1421

age cost and its standard deviation in the *cost_statistics* table. Among the join operations of a multiway join query, the one with the smallest average cost is chosen first. If the range of the selected join operation overlaps with the remaining join operations, they are traced together. As k becomes larger, the range of each join operation becomes broader. This means that the number of traced subplans increases, thereby improving the optimality of the optimized plan of a query. However, the time to produce the optimized plan clearly increases as well. By flexibly controlling the number of simultaneously traced subplans, our proposed scheme provides a desirable trade-off between the optimality of an execution plan and the time to generate the plan. Depending on the current processing load, the range variable k can be flexibly adjusted.

The remainder of this paper is organized as follows: In Sect. 2, related work is presented. In Sect. 3, the join cost model is described on the basis of the cost statistics for a multiway join query. In Sect. 4, the proposed method that can find the optimal execution plan of a multiway join query is introduced. In Sect. 5, the performance of the proposed method is analyzed through a series of experiments. Finally, In Sect. 6, our conclusions are presented.

2. Related Work

There are two common techniques for processing a join operator. The first technique, involving a pipelined processing tree (PPT) [17], uses a multiway nested loop to generate the result of a multiway join query without producing any intermediate join result. The second technique uses a binary linear processing tree (BLPT) [17] that gives intermediate join results. A representative research on the first technique is a multi-stream join algorithm called MJoin [1]. It produces the result of a multiway join query without any tree-structured plan. In [4], a greedy-based optimization method is proposed to determine the evaluation order of each operand of an MJoin operator.

STREAM [5] uses the A-Greedy algorithm to adaptively determine the evaluation order of an MJoin operator to minimize the processing cost of a multiway join query whose join selectivities vary unpredictably over time [14], [15]. The ongoing change in the join selectivity of every possible execution order is continuously monitored to adaptively change the evaluation order of each operand in an MJoin operator. In [18], A-Caching focuses specifically on the problem of the adaptive placement and removal of caches in an MJoin operator to optimize join performance. It provides algorithms for selecting caches, monitoring their cost and benefits in current conditions, allocating memory to caches, and adapting as conditions change.

An 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 of the tuples to reflect the dynamically changing runtime selectivity of each query condition. Based on

the eddy query processing framework, a continuously adaptive, continuous query (CACQ) [8] is implemented. It provides a grouped-filter index to simultaneously evaluate multiple selection predicates and a space-efficient generalization of double-pipelined hash joins called SteMs [16]. The execution order of multiple join operations in a multiway join query is dynamically determined by a ticket-based routing policy that executes a lottery between query operators. The winning chances of a particular operator are proportional to the number of tickets it owns. The operator with a higher selectivity gets more tickets, which means the operator will receive more tuples early through the eddy [11].

In [7], as a modified architecture of the original eddy that uses SteMs, a query operator called a STAIR (Storage, Transformation and Access for Intermediate Results) is introduced to perform a join operation. A STAIR holds the runtime state of join operations. It employs the second technique, BLPT. Even though it solves the drawbacks of the original eddy, they address the problems caused by allowing the query state to be modified and migrated across STAIRs during query execution. STAIRs also solve the problem in which the ability of an eddy to adapt is constrained by the state that accumulates in the query operators. On the basis of the cumulative selectivity of every subplan of join operations, a greedy optimization strategy is used to determine the execution plan of STAIRs dynamically.

3. Cost Statistics

To define the cost of a binary join operation, we use a unit-time-based cost model [13] that defines the evaluation cost of a binary join operation to be the processing cost of all incoming tuples in a unit time. In the execution of a binary linear processing tree, the result of one binary join operation should be materialized because it can be one of the operand for the next binary join operation. Consequently, the result size of a binary join operation affects the overall evaluation cost of a multiway join query, so that the unit-time-based cost model is modified for a multiway join query in this paper. The terms used in the modified cost model are described in Table 1. A join operation $R \bowtie S$ should be executed when a new tuple arrives from any of its two operand streams R and S . Accordingly, the evaluation cost $C_i(R \bowtie S)$ of a binary join operation $R \bowtie S$ for newly arriving tuples in unit time should be symmetrically defined as follows.

$$C_i(R \bowtie S) = (\lambda_R + \lambda_S) \times \left(\frac{W_S \times \lambda_S}{|S|} + \frac{W_R \times \lambda_R}{|R|} + 1 \right). \quad (1)$$

The generating cost of the result of a binary join operation, $C_s(R \bowtie S)$ is proportional to the size of the result and is defined as.

$$C_s(R \bowtie S) = \lambda_R \lambda_S (W_R + W_S) \sigma_{R,S}. \quad (2)$$

Therefore the total evaluation cost of a binary join operation on data streams R and S becomes

Table 1 Terms in the cost model.

λ_R	Input rate of stream R
W_R	Window size of stream R
$ R $	Hash table size of stream R
ΔR	New tuple s inserted into stream R
$\sigma_{R,S}$	Join selectivity of $R \bowtie S$
$C_i(R \bowtie S)$	Join evaluation cost per unit time
$C_s(R \bowtie S)$	Result processing cost per unit time
$C(R \bowtie S)$	Total cost of $R \bowtie S$

$$C(R \bowtie S) = C_i(R \bowtie S) + C_s(R \bowtie S). \quad (3)$$

A multiway join query can be evaluated using a number of different execution plans. All combinations of the join operations that can appear in the plans are maintained in *cost_statistics*. A row in *cost_statistics* is formally defined as follows.

Definition 1. (*Cost_Statistics*) Given a multiway join query with n join operations $J_q = \{J_1, \dots, J_n\}$, the costs of all the join operations ever considered are stored in the *cost_statistics* (CS) table. Each row of the CS table maintains the statistics $CS^i(J_q^m, \mu(C(J_q^m)), \sigma(C(J_q^m)), \text{count}(J_q^m))$, where J_q^m denotes a join operation whose result is a subset of J_q ($J_q^m \subset J_q$), $\mu(C(J_q^m))$ and $\sigma(C(J_q^m))$ denote the average cost of the J_q^m and its standard deviation. The term $\text{count}(J_q^m)$ denotes how many times the join operation J_q has been executed before. If $\text{count}(J_q^m) = 0$, the join operation J_q^m has not been executed before, so that its current statistics are estimated.

When a join operation is not executed, the cost of the join operation is estimated using Eq. (3), and the estimated selectivity is reflected to its statistics in the CS table. When the join operation is part of the optimized execution plan, its real selectivity is obtained while executing the plan and is reflected onto the CS table. If this is the first real selectivity of the join operation, its past estimated statistics are deleted and its statistics are replaced with the real selectivity. In addition, its count field is set to 1. The statistics of a join operation in the CS table represent how the cost of the join operation fluctuates. If the cost of a join operation fluctuates widely in the past, its standard deviation becomes large.

Example 1 Consider the following simple SQL-like query Q that contains three join predicates:

```
Q: SELECT R.a, S.b, T.c
FROM R[10 sec], S[10 sec], T[10 sec], U[10 sec]
WHERE R.a = S.a and S.b = T.b and T.c = U.c
```

This is a simple three-join operation between the streams R , S , T and U . Table 2 shows the *cost_statistics* table of the query Q when the execution plan $(R \bowtie S) \bowtie (T \bowtie U)$, has been executed three times and the other plan $((S \bowtie T) \bowtie R) \bowtie U$ has been executed five times before.

Table 2 Cost_statistics table of the query Q.

Join Op	id	μ	σ	count
$R \bowtie_a S$	J_1	5000	400	3
$S \bowtie_b T$	J_2	4500	200	5
$T \bowtie_c U$	J_3	6200	100	3
$R \bowtie_a S \bowtie_b T$	J_4	2000	150	0
$R \bowtie_a S \bowtie_c U$	J_5	4300	200	5
$S \bowtie_b T \bowtie_c U$	J_6	2300	100	5
$R \bowtie_a S \bowtie_b T \bowtie_c U$	J_7	6200	120	0
$R \bowtie_a S \bowtie_c U \bowtie_b T$	J_8	1800	80	5
$R \bowtie_a S \bowtie_c U \bowtie_b T$	J_9	2700	200	0
$R \bowtie_a S \bowtie_b T \bowtie_c U$	J_{10}	8200	320	3

The values of μ and σ are merely example values.

4. k-Extended Greedy Algorithm

4.1 Subplan and Set of Traced Subplans (STSs)

A complete execution plan for a multiway join query with n join operations is a binary linear processing tree that determines the execution order of n join operations. The objective of query optimization is to find the execution order with the smallest cost. The query can be evaluated using a number of different execution plans. To denote part of an execution plan, the term *m*-subplan is defined formally in Definition 2.

Definition 2. (*m-subplan*) Given a multiway join query q with n join operations $J_q = \{J_1, \dots, J_n\}$, an *m-subplan* of q $sp^m(q)$ is a partial execution plan with m join operations J_q^m ($J_q^m \subset J_q$, $m \leq n$) with the following properties:

- 1) It is a forest of binary linear processing trees and the level of all the join operations in the subplan are less than or equal to m .
- 2) There are $(n-m)$ possible child subplans, each of which is $sp^m(q) \cup \{J_e\}$, where $J_e \notin J_q^m$ and $J_e \in J_q$.

The evaluation cost of each subplan is defined by the accumulated cost of all the join operations in the subplan. Since the cost of an individual join operation is maintained by the average cost and its standard deviation in the CS table, the cost of the subplan is also represented in the same manner as follows:

Definition 3. (*Cost of m-subplan*) Given a multiway join query q with n join operations $J_q = \{J_1, \dots, J_n\}$, its child plan sp^{m+1} is $sp^m \cup \{J_e\}$, whose cost is obtained as follows:

- i) The average cost of sp^{m+1} , $\mu(sp^{m+1}) = \mu(sp^m) + \mu(J_e)$.
- ii) Suppose that sp^m and J_e are independent, the standard deviation of sp^{m+1} , denoted by $\sigma(sp^{m+1})$, is $\sqrt{\sigma(sp^m)^2 + \sigma(J_e)^2 + 2\text{cov}(sp^m, J_e)}$. $\text{cov}(sp^m, J_e)$ is a covariance, which is a measure of how much two variables, $\sigma(sp^m)$ and $\sigma(J_e)$, change together. Since sp^{m-1} and J_e are independent, $\text{cov}(sp^m, J_e) = 0$, so that $\sigma(sp^{m+1})$ is $\sqrt{\sigma(sp^m)^2 + \sigma(J_e)^2}$.

The effective range of an m -subplan is defined by a range variable k in *Definition 4*.

Definition 4. (*Effective range of subplan*) Given a user-defined range variable k , the effective range of sp^m is represented by the range $[\mu(sp^m) - k\sigma(sp^m), \mu(sp^m) + k\sigma(sp^m)]$. The minimum value of the range, i.e., $\mu(sp^m) - k\sigma(sp^m)$, is denoted by $\hat{C}_{min}(sp^m)$ and the maximum value of the range, i.e., $\mu(sp^m) + k\sigma(sp^m)$, is $\hat{C}_{max}(sp^m)$.

As mentioned above, k-EGA simultaneously traces a set of promising subplans, which is defined by a set of traced subplans (STS) as follows.

Definition 5. (*Set of traced subplan, STS^m*) Given a range variable k for a multiway join query q , let SP^m be a set of all the m -subplans of q and sp_{min}^m be the m -subplan with the minimum average cost ($sp_{min}^m \in SP^m$). Among all the possible m -subplans of the query, the elements of STS^m satisfy

$$STS^m = \{sp_i^m \mid sp_i^m \in SP^m, \hat{C}_{min}(sp_i^m) < \hat{C}_{max}(sp_{min}^m)\}.$$

The standard deviation of the cost of a subplan represents how much the cost of the subplan fluctuates. When the value is small, the cost is almost the same in the past, which indicates the cost can accurately represent the current cost of the subplan. In contrast, when the value is large, the cost estimation based on the average cost is not accurate enough. Our scheme traces not only the subplan with the smallest average cost, but also those subplans whose effective ranges overlap with the subplan together. The subplan whose cost cannot be estimated exactly is most likely to be traced because the effective ranges of the subplan are wide. Consequently, the subplan is examined more closely. This is because one of the subplans may lead to a better plan.

Intuitively, the range variable k affects the cardinality of its STS^m . The greater the value is, the wider the effective range of each subplan becomes. Consequently, the size of STS^m increases as well. Furthermore, it means that more promising subplans are traced simultaneously to find out the final optimized execution plan. Therefore, the optimality of the final plan would be better than the one obtained by the general greedy method. On the other hand, the time to generate the final plan may be increased. The proposed method provides a good trade-off between the optimality of the final plan and the time to generate the final plan since they can be flexibly controlled by the range variable k .

Figure 1 illustrates how a range variable k plays a role in controlling the cardinality of STS . Suppose that there are four m -subplans $\{sp_1^m, sp_2^m, sp_3^m, sp_4^m\}$ and that they are the candidate m -subplans for a multiway join query with n join operations ($m \leq n$). To determine the STS^m of the query, the subplan with the smallest average cost is chosen first as in the greedy optimization strategy. Consequently, the subplan sp_3^m is selected first. The effective range of each of these four subplans is calculated. Suppose that the range variable k is set to 1, then the effective range of sp_3^m be-

Fig. 1 Controlling the cardinality of STS^m .

comes $[150 - 50, 150 + 50]$. Similarly, the effective ranges of the remaining subplans can be found. Since the maximum effective range of sp_{min}^m (sp_3^m) is 200, those subplans whose minimum value of the effective range $\hat{C}_{min}(sp^m)$ is less than or equal to 200 are inserted into STS^m as well. As a result, STS^m becomes $\{sp_1^m, sp_3^m\}$. If k is 2, STS^m is $\{sp_1^m, sp_2^m, sp_3^m\}$. Consequently, although the greedy optimization algorithm only traces the m -subplan sp_3^m , the proposed method traces the subplan sp_1^m additionally.

4.2 k-Extended Greedy Algorithm

The overall steps of k-EGA are very similar to those in the greedy optimization strategy that determines the execution order of one of unordered join operations in each optimization step. Likewise, given a multiway join query q with n join operations $J_q = \{J_1, \dots, J_n\}$, k-EGA determines STS^1 in the first step and then generates its child subplans STS^2 . This procedure is continued until STS^n is found. Among the n -subplans in STS^n , that with the minimum evaluation cost sp_{min}^n ($sp_{min}^n \in STS^n$) is the final optimized plan. Given $STS^m = \{sp_1^m, sp_2^m, \dots, sp_k^m\}$, STS^{m+1} is generated. For each m -subplan sp_i^m ($1 \leq i \leq k$), one additional join operation J which is not part of sp_i^m is added to sp_i^m to generate a candidate $(m+1)$ -subplan. Although there are $(n-m)$ different $(m+1)$ -subplans for each sp_i^m , only those $(m+1)$ -subplans that satisfy Definition 5 are inserted into STS^{m+1} . In order to determine sp_{min}^{m+1} , all the subplans generated by STS^m should be scanned after they are generated. However, maintaining all the child subplans of STS^m is a runtime overhead. sp_{min}^{m+1} can be incrementally identified by tracing each of m -subplans in STS^m . When sp_{min}^m is changed, the $(m+1)$ -subplans already inserted into STS^{m+1} are re-examined. The pseudo-code of these steps is shown in Fig. 3. The following example shows how k-EGA finds the optimal plan of a multiway join query.

Example 2) Suppose that there are the SQL-like query Q in example 1 and the *cost_statistics* in Table 2. Let k be 1.

1) (*Determine STS^1*) Given $SP^1 = \{sp_1^1, sp_2^1, sp_3^1\}$ where $sp_1^1 = \{J_1\}$, $sp_2^1 = \{J_2\}$ and $sp_3^1 = \{J_3\}$. Since $\mu(sp_2^1)$ is the smallest, the subplan becomes sp_{min}^1 by Definition 4. Subsequently, STS^1 becomes $\{sp_1^1, sp_2^1\}$, as shown in Fig. 2 (a), because only sp_1^1 overlaps with sp_2^1 .

2) (*Determine STS^2*) The subplans sp_2^1 and sp_2^2 are gen-

Fig. 2 Steps in generating STS^2 from STS^1 in Example 2.

Algorithm Determine_STS(m, k, sp^{m-1}_m, st_ctl)

```

/*  $m$ : number of join operations in  $sp^m_i$  ( $sp^m_i \in STS^m, m \leq n$ ) */
/*  $k$ : a range variable */
/*  $sp^{m-1}_n$ : subplan needed for making  $SP^m$  */
/*  $st\_ctl$ : cost_statistics that stores stream information */
 $i = 0$ ;  $s = \text{size of } STS^m$ ;  $sp_{min} = \text{null}$ ;
For each  $ctl[i] \in st\_ctl$  do /*  $ctl[i]$  is the  $i^{th}$  element of  $st\_ctl$  */
  If ( $sp^{m-1}_n \cup ctl[i]$  can be  $STS^m$ ) then
    If  $sp_{min}$  is empty then
      Insert  $\{sp^{m-1}_n \cup ctl[i]\}$  into  $sp_{min}$ 
     $s = s + 1$ 
    Insert  $sp_{min}$  into  $STS^m_s$ 
  Else
    If  $\mu(sp_{min}) < \mu(sp^{m-1}_n \cup ctl[i])$  then
      If  $C_{max}(sp_{min}) > C_{min}(sp^{m-1}_n \cup ctl[i])$  then
         $s = s + 1$ 
        Insert ( $sp^{m-1}_n \cup ctl[i]$ ) into  $STS^m$ 
      Else /*  $ctl[i]$  cannot be the element of  $STS^m$  */
        /* Nothing */
      End if
    Else /*  $\mu(sp_{min}) > \mu(sp^{m-1}_n \cup ctl[i])$  */
      Change the  $sp_{min}$  into  $sp^{m-1}_n \cup ctl[i]$ 
      For each  $sp^m_j \in STS^m, 1 \leq j \leq s$  do
        If  $C_{max}(sp_{min}) < C_{min}(sp^m_j)$  then
          Delete  $STS^m_j$ 
         $s = s - 1$ 
      End if
       $s = s + 1$ 
      Insert  $sp_{min}$  into  $STS^m$ 
    End for
  End if
End for
End if
End if
End if
End for

```

Fig. 3 Determine_STS.

erated by sp^1_1 . Likewise sp^2_3 and sp^2_4 are generated by sp^1_2 . The effective ranges of the subplans are presented in Fig. 2 (b). As is shown in Fig. 2 (b), sp^2_4 is sp^m_{min} and sp^2_2

overlaps with sp^2_4 . Therefore, STS^2 becomes $\{sp^2_2, sp^2_4\}$ where $sp^2_2 = \{J_1, J_4\}$ and $sp^2_4 = \{J_2, J_5\}$.

3) The candidates of STS^3 are sp^3_1 and sp^3_2 , where sp^3_1 is $\{J_1, J_4, J_8\}$ and sp^3_2 is $\{J_2, J_6, J_9\}$. $\mu(sp^3_1)$ is lower than $\mu(sp^3_2)$, so that the final query plan becomes $sp^3_1 = \{J_1, J_4, J_8\}$, i.e., $((R \bowtie S) \bowtie T) \bowtie U$.

The standard deviation of a subplan presents how much the subplan fluctuates. The small standard deviation means there is little change in the cost of the subplan. On the other hand, the large standard deviation means the cost of the subplan fluctuated widely in the past, and it is difficult to determine the current cost of the plan. Also, if any subplans fluctuate widely, the following subplan is also likely to be done because the result of the parent can be the input of the child. Therefore, k-EGA traces these subplans more deeply and intensively and this can cause a more optimal plan. (By Definitions 4 and 5, the subplans which have a large standard deviation are likely to be included in promising subplans.)

As mentioned above, the number of simultaneously traced subplans is flexibly controlled. Actually, the capability of the k-EGA is from the general greedy optimization to exhausted optimization. If the range variable k is set to 0, only one subplan is traced since the minimum and maximum values of effective ranges of all subplans become the same by Definition 4. Consequently, it produces the same plan as the general greedy optimization. In contrast, if the range variable k is sufficiently large, so that all the m -subplans can be included in STS^m , its resulting plan is the completely optimized plan.

5. Experiments

In this section, we provide the experimental results of the proposed algorithm. All the algorithms are implemented in C, and all the experiments are executed on a Pentium 4 CPU 2.66 GHz system with 1 GB RAM. The system runs Linux with a 2.4.5 kernel and gcc 3.3.2. To validate the cost model

proposed in this paper, a multiway join query with 2 join operations, $A \bowtie B$ and $B \bowtie C$, is experimented on for a synthetic dataset containing tuples of integer attributes. The domain size of each join attribute is 100. The value of each attribute ranges over $[0..99]$ and is randomly selected. The input rates of the operand data streams A , B and C are 10 (tuples/s), 20 (tuples/s) and 50 (tuples/s), respectively. The cost of the query is compared with the actual execution time of the query by varying either the input rate of the operand A and the domain size of the join attribute of $A \bowtie B$. Figure 4 shows that the cost is closely proportional to the actual execution time.

To analyze the performance of the k -EGA, a synthetic dataset is generated. For a multiway join query, the sizes of a sliding window and a hash table are fixed. The rates of the operand data streams are uniformly chosen from 10 tuples/s to 1000 tuples/s and the domain size of the attribute of each join operation is also randomly varied from 1 to 1000. The value of each join attribute is randomly selected from 1 to the domain size of the join attribute. Figure 5 shows the effects of a range variable k as the number of operand data streams for a multiway join query is increased. If the range variable k is set to 0, k -EGA is the same as the routing policy for a STAIR [7], which routes an incoming tuple to the join operation with the lowest selectivity. Intuitively, the execution time should increase as the number of join operations in a query increases. However, as k is increased, the execution time decreases since the final plan produced by the k -EGA becomes more efficient.

Fig. 4 Comparison of the cost model and the actual execution time of a query plan.

In Figs. 6 and 7, a multiway join query with six join operations is experimented on for the nine synthetic datasets in Table 3 to compare the performance of k -EGA. The input rate and domain size of an operand data stream are randomly selected within the predefined range. The join selectivity of a join operation is determined by the reciprocal of the domain size of the attribute in each join predicate. In Table 3, the term $a \pm b$ means that a value is randomly generated from the range $[a - b, a + b]$. For example, the input rates of the cost in DS_5 are randomly generated from the range $[50, 150]$ and the domain size of each join attribute is randomly generated from the range $[500, 1500]$.

Given the k for a multiway join query, let $Cost_k$ denote the cost of the execution plan generated by k -EGA and $Cost_{opt}$ denote the cost of the optimal plan. To observe the performance of k -EGA with respect to the range variable k , we define the relative effectiveness $E(k)$ of k -EGA. It is defined as.

$$E(k) = \frac{Cost_k - Cost_{opt}}{Cost_{opt}}.$$

The smaller the $E(k)$ is, the closer to the optimal plan the execution plan generated by k -EGA is. In Fig. 6, $E(k)$ decreases as k increases even if the optimality of k -EGA decreases as the range of the input rates becomes wider. As shown in Fig. 7, $E(k)$ decreases as k logarithmically increases even if the optimality of our algorithm decreases as the range of the domain size increases.

To reoptimize the current execution plan continuously, it is important to monitor its cost periodically. If the plan is no longer efficient, k -EGA generates a newly optimized plan. Given the current plan p_i , let $Cost^{latest}(p_i)$ denote the cost of p_i when the plan was originally generated. On the other hand, let $Cost^{cur}(p_i)$ denote the cost of p_i at present. A newly optimized plan should be generated by k -EGA when the following equation is satisfied.

$$\left| \frac{Cost^{latest}(p_i) - Cost^{cur}(p_i)}{Cost^{latest}(p_i)} \right| \geq \delta$$

A cost allowable ratio δ is a user-defined parameter that denotes the allowable ratio of the current cost relatively to the latest optimized cost. As δ is set to be smaller, k -EGA changes its optimized plan more sensitively.

Figure 8 shows how k -EGA adaptively reoptimizes its

Fig. 5 The effects of a range variable k by varying the number of join operations.

Fig. 6 Performance comparison by varying range of input rate.

Fig. 7 Effects of domain size of join attribute.

Table 3 Characteristics of synthetic datasets.

Dataset	Range of the input rates	Domain Size of a join attribute
DS_1	[90, 110]	[900, 1100]
DS_2	[90, 110]	[500, 1500]
DS_3	[90, 110]	[1, 2000]
DS_4	[50, 150]	[900, 1100]
DS_5	[50, 150]	[500, 1500]
DS_6	[50, 150]	[1, 2000]
DS_7	[1, 200]	[900, 1100]
DS_8	[1, 200]	[500, 1500]
DS_9	[1, 200]	[1, 2000]

Table 4 Sub-datasets for Fig. 8.

	Data Size (tuples in s_i)	input rate	selectivity for $s_1 \sim s_4$	selectivity for $s_5 \sim s_7$
D_1	5,000	$s_1 \sim s_7: 100$	0.0001	0.001
D_2	5,000	$s_1 \sim s_7: 100$	0.001	0.0001
D_3	5,000	$s_1 \sim s_4: 100$ $s_5 \sim s_7: 1000$	0.001	0.0001

Fig. 8 Adaptivity of k -EGA.

execution plan for a seven-way join query. To simulate the dynamic change in the characteristics of input data streams, the tuples of the three datasets D_1 , D_2 and D_3 in Table 4

are processed in sequence. The y-axis in Fig. 8 shows the cost per 1000 input tuples. The term “Optimal for D_1 ” denotes the static optimal plan for the dataset D_1 only. When the tuples in D_2 start being processed, the cost of the optimal plan for D_1 sharply increases since the current evaluation sequence is no longer optimal to D_2 . However, k -EGA ($k = 1$) reoptimizes to generate a newly adjusted plan, and its execution plan cost remains to be close to the cost of executing the optimal plan.

Figure 9 shows the trade-off between the optimality of the final plan generated by k -EGA and the time to generate the final plan. When $k < 2$, the cost of the execution plan optimized by k -EGA markedly decreases while the optimization time almost linearly increases. However, when k is large, the time to generate the final plan exponentially increases while the cost of the plan almost does not decrease. This means that when k becomes larger than a specific threshold (in this experiment, it is 2), the overhead of tracing multiple subplans simultaneously becomes greater than the profits gain obtained by finding a better plan.

Fig. 9 Trade-Off between the optimality and the optimization time ($n = 7$).

6. Conclusions

In this paper, to optimize a multiway join query in a data stream environment, the K-Extended Greedy Algorithm (k-EGA) is proposed. The proposed algorithm traces a set of promising subplans simultaneously to reduce the possibility of producing a suboptimal plan. We provide a method of controlling the number of simultaneously traced subplans flexibly with respect to the range variable k . As a result, it is possible to adaptively control the number of subplans by changing the range variable k depending on the current processing load. Experiment results show how efficiently k-EGA can optimize a multiway join query in various stream environments. k-EGA can provide a good trade-off between the optimality of a generated execution plan and its optimization time. In addition, the plan generated by an MJoin operator cannot be utilized in a multiple query operation since the join orders of all operand streams are different. However, it is easy to extract common join operations in a number of plans generated by k-EGA. Most of the continuous queries are registered in advanced and should be processed continuously, so multiple query optimization is very important in a data stream environment.

Acknowledgments

"This work was supported by the Korea Science and Engineering Foundation (KOSEF) NRL Program (No. R0A-2006-000-10225-0) and was partially supported by from the Basic Research Program (No. R01-2006-000-11223-0) grant funded by the Korea government (MEST)."

References

- [1] S. Viglas, J.F. Naughton, and J. Burger, "Maximizing the output rate of multiway join queries over streaming information sources," VLDB 2003, pp.285–296, 2003.
- [2] S. Viglas and J.F. Naughton, "Rate-based query optimization for streaming information sources," SIGMOD 2002, pp.37–48, 2002.
- [3] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," SIGMOD 2003, pp.40–51, 2003.
- [4] L. Golab and M.T. Özsu, "Processing sliding window multiway joins in continuous queries over data streams, sources," VLDB 2003, pp.500–511, 2003.

- [5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive ordering of pipelined stream filters," SIGMOD 2004, pp.407–418, 2004.
- [6] M.A. Hammad, W.G. Aref, and A.K. Elmagarmid, "Stream window join: Tracking moving objects in sensor-network databases," SSDBM 2003, pp.75–84, 2003.
- [7] A. Deshpande and J.M. Hellerstein, "Lifting the burden of history from adaptive query processing," VLDB 2004, pp.948–959, 2004.
- [8] S. Madden, M. Shah, J.M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," SIGMOD 2002, pp.49–60, 2002.
- [9] B. Gedik, K.-L. Wu, P.S. Yu, and L. Liu, "GRUBJOIN: An adaptive multi-way windowed stream join with time correlation-aware CPU load shedding," TKDE, vol.19, no.10, pp.1363–1380, Oct. 2007.
- [10] A.M. Ayad and J.F. Naughton, "Static optimization of conjunctive queries with sliding windows over infinite streams," SIGMOD 2004, pp.419–430, 2004.
- [11] R. Avnur and J. Hellerstein, "Eddies: Continuously adaptive query processing," ACM SIGMOD Record, vol.29, no.2, pp.261–272, June 2000.
- [12] A. Deshpande, "An initial study of overheads of eddies," ACM SIGMOD Record, vol.33, no.1, pp.44–49, March 2004.
- [13] J. Kang, J.F. Naughton, and S.D. Viglas, "Evaluating window joins over unbounded streams," ICDE 2003, pp.341–352, 2003.
- [14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," PODS 2002, pp.1–16, 2002.
- [15] L. Golab and M.T. Özsu, "Issues in data stream management," ACM SIGMOD Record, vol.32, no.2, pp.5–14, June 2003.
- [16] V. Raman, A. Deshpande, and J.M. Hellerstein, "Using state modules for adaptive query processing," ICDE 2003, pp.353–364, 2003.
- [17] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of non-recursive queries," VLDB 1986, pp.128–137, 1986.
- [18] S. Babu, K. Munagala, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," ICDE 2005, pp.118–129, 2005.

Hong Kyu Park received the B.S. and M.S. degree in Computer science from Yonsei University, Seoul, Korea, in 2004 and 2007. He is currently a Ph.D. candidate at the Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include data stream processing and data stream management systems.

Won Suk Lee received the B.S. degree in Computer Engineering from Boston University, Boston, MA and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from Purdue University, West Lafayette, IN. He was a senior staff engineer at Computer Division, Samsung Electronics. He is currently a professor of Department of Computer Science at Yonsei University, Korea. His current research interests include data stream mining, data stream management systems, and real-time OLAP.