Multi-Join Continuous Query Optimization: Covering the Spectrum of Linear, Acyclic, and Cyclic Queries

Venkatesh Raghavan, Yali Zhu, Elke A. Rundensteiner, and Daniel Dougherty

Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609, USA
{venky, yaliz, rundenst, dd}@cs.wpi.edu

Abstract. Traditional optimization algorithms that guarantee optimal plans have exponential time complexity and are thus not viable in streaming contexts. Continuous query optimizers commonly adopt heuristic techniques such as Adaptive Greedy to attain polynomial-time execution. However, these techniques are known to produce optimal plans only for linear and star shaped join queries. Motivated by the prevalence of acyclic, cyclic and even complete query shapes in stream applications, we conduct an extensive experimental study of the behavior of the state-of-the-art algorithms. This study has revealed that heuristic-based techniques tend to generate sub-standard plans even for simple acyclic join queries. For general acyclic join queries we extend the classical IK approach to the streaming context to define an algorithm *TreeOpt* that is guaranteed to find an optimal plan in polynomial time. For the case of cyclic queries, for which finding optimal plans is known to be NP-complete, we present an algorithm *FAB* which improves other heuristic-based techniques by (i) increasing the likelihood of finding an optimal plan and (ii) improving the effectiveness of finding a near-optimal plan when an optimal plan cannot be found in polynomial time. To handle the entire spectrum of query shapes from acyclic to cyclic we propose a *Q-Aware* approach that selects the optimization algorithm used for generating the join order, based on the shape of the query.

1 Introduction

1.1 Continuous Query Plan Generation

In traditional, static, databases, query optimization techniques can be classified as either techniques that generate optimal query plans [1–4], or heuristic based algorithms [5–7], which produce a good plan in polynomial time. In recent years there has been a growing interest in continuous stream processing [8–11]. Continuous query processing differs from its static counterpart in several aspects. First, the incoming streaming data is unbounded and the query life span is potentially infinite. Therefore, if state-intensive query operations such as joins are not ordered correctly they risk consuming all resources. Second, live-stream applications such as fire detection and stock market tickers are time-sensitive, and older tuples are of less importance. In such applications, the query execution must keep up with incoming tuples and produce real-time results at an optimal output rate [12]. Third, data stream statistics typically utilized to generate the execution query plan such as input rates, join selectivity and data distributions, will change over time. This may eventually make the current query plan unacceptable at some point of the query life span. This volatile nature of the streaming environment makes re-optimization a necessity. We conclude that an effective continuous query optimizer must have (1) polynomial time complexity, (2) the ability to generate optimal or at least near-optimal plans, (3) provide migration strategies to move the existing plan states to the newly generated plan. Existing approaches either have exponential complexity [1, 3, 4] or fail

to produce an optimal plan. In streaming databases, the typical approach [9, 10, 13, 14] is to use a *forward greedy heuristic* [3]. Although it was shown in [15] that a greedy approach can perform within a constant factor of optimal, the scope of this analysis was restricted to ordering unary filters over a single stream. Migration strategies proposed by [16] to safely transfer the current suboptimal query plan to the re-optimized query plan can be employed.
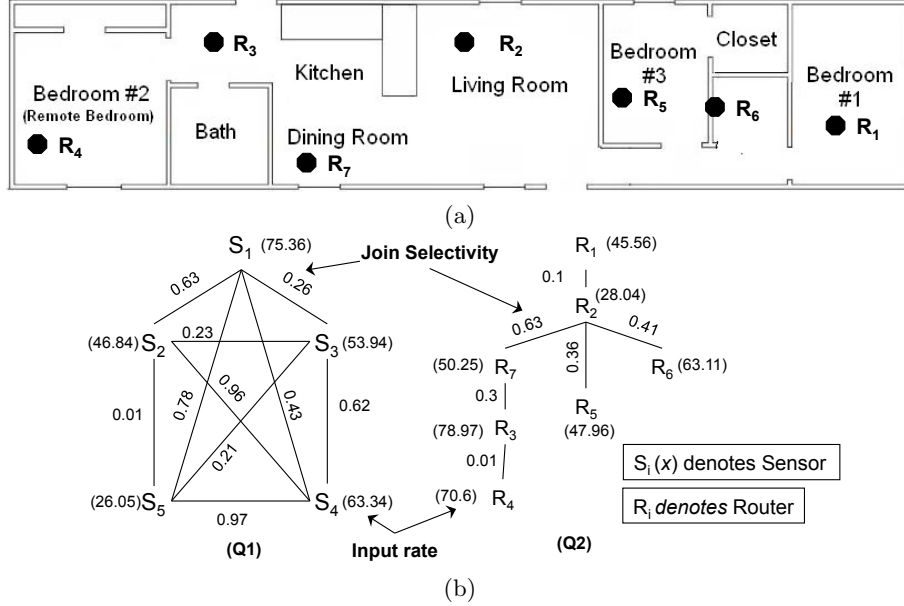

Fig. 1. a) NIST Fire Lab - Mobile Home Test Arena, b) Join Graphs for $Q1$ and $Q2$

1.2 Spectrum of Linear, Acyclic and Cyclic Queries

In any streaming domain many flavors of join graphs from cyclic to acyclic may coexist. As motivation we examine the processing of sensor readings obtained from sensors placed in the mobile home (Figure 1.a) conducted by National Institute of Standards and Technology (NIST) [17]. Each sensor generates a reading feed S_i made up of tuples that contain the sensor identifier *sid*, the time stamp of the reading and the actual reading. In addition, each room contains a router (R_i) that generates a summarizing stream of all the sensors in its respective room. Fire engineers interested in determining false positive alarms or rogue sensors can submit the following query:

```
Query 1 (Q1): A smoke sensor in the bedroom #1 has detected an abnormality.
Monitor and compare its behavior with all sensors within the same room.
```

To evaluate $Q1$ we must compare the readings of each sensor against the readings of all the sensors in bedroom #1 to identify abnormalities. Such user queries are represented as complete (cyclic) join graphs, as in Figure 1.b. In the same domain, user queries can also correspond to acyclic join graphs. For example, first responders must identify the fire and smoke path in an arena.

```
Query 2 (Q2): Find the direction and the speed of the smoke cloud caused
by the fire generated in bedroom #1.
```

The spread of fire and smoke is guided by access paths such as doors and vents. Such queries can be answered by querying the summarized streams from the routers placed in each room. Query $Q2$ is therefore guided by the building layout, which in Figure 1.b is an acyclic join graph.

1.3 Our Contributions

The optimization of cyclic join queries is known to be NP-complete [5]. Even in the case of acyclic join queries, the state-of-the-art techniques [15, 18, 19] used in streaming database systems do not guarantee optimality. In addition, these techniques are insensitive to the shape of user query and so represent a "one-algorithm fits-all" policy. There is a need for a comprehensive approach to handle the entire spectrum of join queries, from acyclic to cyclic in the most effective manner.

In this effort, we begin by studying the performance of the commonly adopted heuristic-based techniques [15, 18, 19]. These techniques are known to produce sub-standard query plans for general acyclic and cyclic queries and may prove fatal for time-critical applications. More specifically, when handling acyclic queries our experiments demonstrate several cases when these techniques are shown to produce sub-optimal plans that are 5 fold more expensive than their optimal counterparts. In response, we tackle this shortcoming by extending the classical *IK algorithm* [5] in the streaming context (Section 4). The resulting *TreeOpt* approach while still featuring polynomial time complexity now also guarantees to produce optimal join plans for acyclic continuous join queries, such as $Q2$.

Subsequently, we focus on query plans represented by cyclic join graphs. Our experiments reveal that when handling cycles, in several cases the popular heuristic-based techniques generate plans that are 15 fold more expensive than their optimal counterpart. Unfortunately, even the adaptation of TreeOpt for cyclic queries is not guaranteed to generate optimal plans. Since the optimization problem in this setting is NP-complete we refine our goals as follows. We ask that our optimizer (i) be polynomial in time complexity, (ii) be able to increase the probability of finding optimal plans, and (iii) in scenarios where an optimal solution cannot be found, the technique should decrease the ratio of how expensive the generated plan is in comparison to the optimal plan. Towards this end, we introduce our *Forward and Backward Greedy* (*FAB*) algorithm that utilizes our *global impact ordering* technique (see Section 5). This can be applied in parallel with the traditional *forward greedy* approach [3]. Through our experimental evaluation we show that our FAB algorithm has a much higher likelihood of finding an optimal plan than state-of-the-art techniques. In scenarios when an optimal plan cannot be found, FAB is shown to generate plans that are closer to the optimal than those generated by current approaches. Finally, we put the above techniques together into a system that is query shape aware while still having a polynomial time complexity, called the *Q-Aware* approach (see Section 6). This technique is equipped to generate the best possible (optimal or a good) plans guided by the shape of the query.

2 Background

2.1 Continuous Multi-Join Processing

The common practice for executing multi-join queries over windowed streams is by a multi-way join operator called mjoin [18], a single multi-way operator that takes as input the continuous streams from all join participants. See Figure 2.a. Two benefits of mjoin are that the order in which the inputs tuples from each stream are joined with remaining streams can be dynamic, and intermediate tuples are not longer stored, saving space. To illustrate, in Figure 2.b new tuples from S_1 (ΔS_1 for short) are first inserted into the state of S_1 (denoted as ST_{S_1}), then used to probe the state of S_4 (ST_{S_4}), and the resulting join tuples then go on to probe ST_{S_2}, etc. This ordering is

called S_1's pipeline. The *join graph* (JG), as in Figure 1.b, represents a multi-join query along with statistical information such as input rates and selectivities. In this work, we assume independent join selectivities and time window constraints [20, 21].
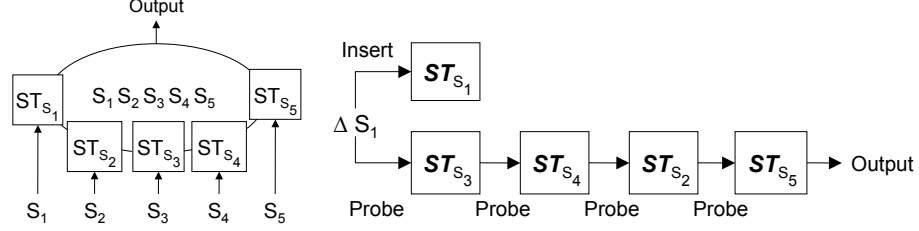

Fig. 2. a) 5-way Mjoin operator b) Sample join ordering to process new tuples from S_1 (ΔS_1)

2.2 Cost Analysis

The cost model is based on the commonly used per-unit-time cost metrics [20]. The estimated processing cost of an mjoin is the cumulative sum of the costs to process updates from each of the n input streams.

Table 1. Terms Used In The Cost Model

Term	Meaning
n	Number of participant streams
$\triangle S_i$	ith update stream
$\bowtie_{S_i:1}, ..., \bowtie_{S_i:n-1}$	Join order (pipeline) for $\triangle S_i$
$\bowtie_{S_i:j}$ or $S_i : j$	jth stream to join in $\triangle S_i$'s pipeline
C_{insert}	Cost of inserting a tuple into a state
C_{delete}	Cost of deleting a tuple into a state
C_{join}	Cost of joining a pair of tuples
$\lambda_{S_i:j}$	Average input rate of stream $S_i : j$
$\sigma_{S_i:j}$	Selectivity for joining stream $S_i : j$
W	Sliding time-based window constraint
$|ST_{S_i}|$	Number of tuples in state of stream S_i

Without loss of generality, we estimate the per-unit-time CPU cost of processing the update from stream S_1. It is the sum of the costs incurred in inserting the new tuples ($insert(S_1)$) to its corresponding state (ST_{S_1}), to purge tuples ($purge(S_1)$) that fall outside the time-frame (W) and to probe the states of the participating streams ($probe(\bowtie_{S_1:1}, \bowtie_{S_1:2}, ..., \bowtie_{S_1:n-1})$).

$$CPU_{S_1} = insert(S_1) + purge(S_1) + probe(\bowtie_{S_1:1}, \bowtie_{S_1:2}, ..., \bowtie_{S_1:n-1}) \qquad (1)$$

The cost for inserting new tuples from stream S_1 into the state ST_{S_1} is $\lambda_{S_1} * C_{insert}$ where C_{insert} is the cost to insert one tuple. Tuples whose time-stamp is less than ($time_{current}$ - W) are purged. Under the uniform arrival rate assumption, the number of tuples that will need to be purged is equivalent to λ_{S_1}. If the cost for deleting a single tuple is given by C_{delete}, then the purging cost for stream S_1 is $\lambda_{S_1} * C_{delete}$. Updates from stream S_1 are joined with the remaining join participants in a particular order as specified by its pipeline $\bowtie_{S_1:1}, \bowtie_{S_1:2}, \ldots, \bowtie_{S_1:n-1}$. The cost of joining every

new tuple from S_1 with $\bowtie_{S_1:1} (= S_3)$ is $\lambda_{S_1} * (\lambda_{\bowtie_{S_1:1}} * W) * C_{join}$. This is due to the fact that under a constant arrival rate at most $(\lambda_{\bowtie_{S_1:1}} * W)$ tuples exist in the state of $\lambda_{S_1:1}$ stream which will join with the new updates from S_1. Now, the resulting tuples $(\lambda_{S_1} * (\lambda_{\bowtie_{S_1,1}} * W) * \sigma_{\bowtie_{S_1:1}})$ probe the state of $\bowtie_{S_1:2}$ and so on. Thus the total update processing cost is:

$$
\begin{aligned}
CPU_{S_1} =& \lambda_{S_1} * C_{insert} + \lambda_{S_1} C_{delete} + \lambda_{S_1} * (\lambda_{\bowtie_{S_1:1}} * W) * C_{join} \\
& + (\lambda_{S_1} * [\lambda_{\bowtie S_1:1} * W] * \sigma_{\bowtie_{S_1:1}}) * (\lambda_{\bowtie_{S_1:2}} * W) * C_{join} + ... \\
=& \lambda_{S_1}[C_{insert} + C_{delete} + (\sum_{i=1}^{n-1}[\prod_{j=1}^{i} \lambda_{\bowtie_{S_1:j}} * \sigma_{\bowtie_{S_1:j-1}}] * W^i * C_{join})]; where \; \sigma_{i:0} = 1.
\end{aligned}
\tag{2}
$$

It follows that the CPU cost for any n-way mjoin is:

$$
CPU_{mjoin} = \sum_{k=1}^{n}(\lambda_{S_k}[C_{insert} + C_{delete} + \sum_{i=1}^{n-1}[\prod_{j=1}^{i} \lambda_{\bowtie_{S_k:j}} \sigma_{\bowtie_{S_k:j-1}}]W^i C_{join})])
\tag{3}
$$

where k is the number of streams in the mjoin, while i and j are the index over the pipeline.

3 Assessment of Popular Optimization Algorithms

3.1 Dynamic Programming Techniques For Ordering MJoin

The classical bottom-up dynamic programming algorithm due to Selinger [1] is guaranteed to find an optimal ordering of *n-way* joins over relational tables, and can easily be adopted for mjoin ordering (DMJoin). The aim of the algorithm to find the join order that produces the least number of intermediate tuples. This is consistent with the CPU cost model described in Equation 3. To illustrate, consider our query $Q1$ (as depicted in Figure 1.b) and the processing of the new tuples from stream S_1. In the first iteration, the algorithm computes the cost of all join subsets with S_1 and one other stream. For example, $\{S_1, S_2\}$, $\{S_1, S_3\}$, etc., are each considered. Next, the algorithm considers all subsets having $k = 3$ streams and S_1 being one of those streams. In the kth iteration $\binom{n-1}{k-1}$ join pairs need to be maintained. For each of these subsets there can be $k - 1$ ways of ordering. Several extensions to this core approach have been proposed [3, 4] along with better pruning techniques [2, 22] have been designed. However, their exponential time complexity makes them not viable for streaming databases.

3.2 Forward Greedy Algorithm

In the streaming context, it is a common practice to adopt some variations of the forward greedy algorithm [3] to order mjoins [15, 18, 19], here called *F-Greedy*. In each iteration, the candidate that generates the smallest number of intermediate tuples is selected as the next stream to be joined.

For $Q1$ (Figure 1.b), in the first iteration the algorithm computes the cost incurred by new tuples from S_1 joining with the remaining streams. For example, stream S_3 generates $75.36 * 53.94 * 0.26 \approx 1057$ tuples/sec, while with stream S_5 is $75.36 * 26.05 * 0.78 \approx 1531$ tuples/sec. Since S_3 produces the smallest number of intermediate results, it is chosen for joining first S_1's pipeline and so on. F-Greedy returns the plan $S_1 \bowtie S_3 \bowtie S_5 \bowtie S_2 \bowtie S_4$ which generates ≈ 7212 intermediate tuples per sec. For comparison, the optimal plan for processing input tuples from S_1 is $S_1 \bowtie S_5 \bowtie S_2 \bowtie S_3 \bowtie S_4$ which generates ≈ 3629 intermediate tuples per second. Therefore, F-Greedy plan is 2 fold more expensive than the optimal plan generated by DMJoin.

Time complexity to generate a query plan that processes the new tuples from an input stream is $\mathcal{O}(n^2)$. Therefore, the time complexity for ordering an *n*-way mjoin operator is $\mathcal{O}(n^3)$.
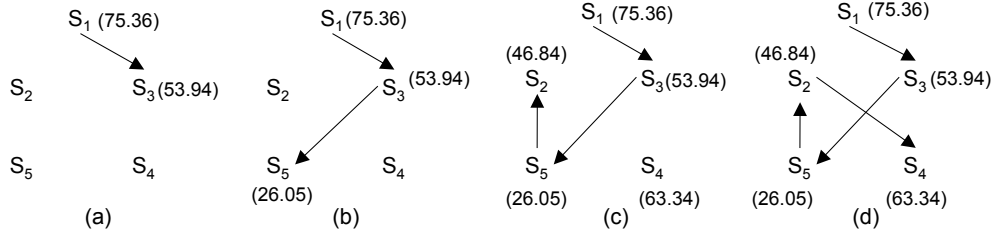
Fig. 3. Forward Greedy Algorithm

3.3 Experimental Evaluation

Environment. Our experiments were conducted on a 3.0 GHz 4 Intel machine with 1 GB memory. We executed the generated plans on the CAPE continuous query engine [10] to verify the cost model and record the execution time. All algorithms where implemented in Java 1.5.

Experimental Setup. In line with state-of-the-art techniques [23, 24] we use synthesized data to control the key characteristics of the streams namely input rate and selectivites between streams. More importantly, to assure statistical significance of our results and scopes of applicability we work with wide variations of settings as elaborated below. The setup included varying the number N of streams from 3 to 20. For each N, we randomly generate: 1) the input rate for each stream [1–100] tuples/sec, and 2) the join selectivities among the streams. For each N, we repeat this setup process 500 times. Therefore we have a total of $(20-3+1) * 500 = 9000$ different parameter settings.
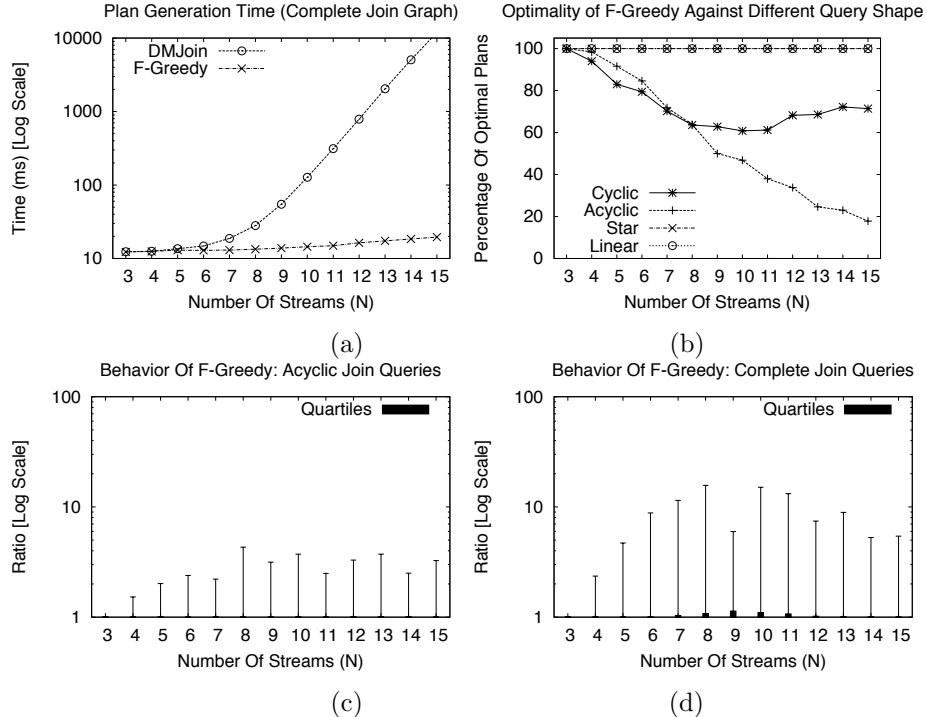

Fig. 4. F-Greedy vs. DMJoin: (a) Plan Generation Time; (b) % of Optimal Plans Generated by F-Greedy; (c) General Acyclic Join Graph; (d) Complete Join Graph

Objectives. First, we compare time needed by the algorithm to generate a plan. Second, we measure the effectiveness measured as % of runs[1] of each algorithm to produce an optimal plan. Third, we compare the plan produced by heuristic-based algorithm against the optimal plan returned by DMJoin. Lastly, we observe the effectiveness of generating optimal plans as well as how expensive non-optimal plans can be for a diversity of join graph shapes.

Evaluation of Popular Algorithms. We begin by comparing the time needed to generate a plan by F-Greedy vs. DMJoin. The plan generation time for each distinct N is the average time over 500 distinct runs. As it is well known, Figure 4.a re-affirms the exponential nature of DMJoin. Next, we study the capability of F-Greedy to generate optimal plans for different query plan shapes. We achieve this by comparing the cost of plans generated by F-Greedy to those generated by DMJoin. As in Figure 4.b F-Greedy generates optimal plans for linear and star-shaped join queries. However, for general acyclic and cyclic join queries, F-Greedy generates substandard plans for many settings. Next, to provide a deeper understanding of the behavior of F-Greedy when applied to general acyclic and cyclic join queries, we compute the ratio of the average number of intermediate results generated by plans produced by heuristic-based algorithm against those generated by optimal plan. A ratio = 1 is ideal since it reflects that F-Greedy plan generates the same number of intermediate results and therefore have similar CPU costs. Figures 4.c and 4.d confirms that F-Greedy could potentially generate plans that are many fold more expensive than the optimal plan, thereby triggering re-optimization sooner.

4 The TreeOpt Algorithm

In the previous section, F-Greedy is shown to generate substandard plans even for acyclic join queries. We now extend the *classical IK algorithm* [5] to solve the optimal ordering of acyclic queries, called *TreeOpt*. [5] has been largely ignored in the streaming context. To illustrate the main intuition of this approach let us consider acyclic join graphs such as $Q2$ (Figure 1.b) and the processing of new tuples from stream S_i. The join graph can now be viewed as a directed tree with S_i as its root. The aim is to transform this tree into a chain (representing the join order) with S_i as its root. However, a directed tree can be composed of vertexes linked as a chain or as wedges (two or more children). If this directed tree is a chain, then the chain hierarchy dictates the join order. When this directed tree is not a chain, then TreeOpt starts the optimization from its leaves. Each vertex (S_r) is assigned a rank (defined later). If a parent vertex S_q has a greater rank than that of its child vertex S_r, then the two vertexes are merged into one with the name S_qS_r and their rank is calculated. The merging of unordered vertexes ensures that the structural information of the query is not lost. For example, S_q is always joined before S_r. To transform a wedge into a chain, we merge all the children of a vertex into a single chain arranged in ascending order by their respective rank.

Next, we show that the cost model, as in Section 2.2, satisfies the Adjacent Sequence Interchange (ASI) property of [5] and thereby is guaranteed to generate an optimal plan for acyclic join graphs. Consider a given acyclic join graph JG and a join sequence $\zeta = (\bowtie_{S_1:0}, ..., \bowtie_{S_1:n-1})$ starting from input S_1. By Equation 2 the total CPU cost of this sequence is: $CPU_{S_1} = \lambda_{S_1} [C_{insert} + C_{delete} + (\sum_{i=1}^{n-1} [\prod_{j=1}^{i} \lambda_{\bowtie_{S_1:j}} * \sigma_{\bowtie_{S_1:j-1}}] * W^i * C_{join})$. The terms $\lambda_{S_1}(C_{insert} + C_{delete})$ and C_{join} are order-independent, and are therefore ignored. The order-dependent part of CPU_{S_1}, that is, $\sum_{i=1}^{n-1} [\prod_{j=1}^{i} \lambda_{\bowtie_{S_1:j}} * \sigma_{\bowtie_{S_1:j-1}}] * W^i$ can be defined recursively as below (similar to [5]):

---

[1] *A run is defined as a unique assignment of input rates and selectivities in a join graph*

$$C(\Lambda) = 0 \qquad\qquad\qquad\qquad \text{Null sequence } \Lambda.$$
$$C(S_1) = 0 \qquad\qquad\qquad\qquad \text{Starting input stream.}$$
$$C(\bowtie_{S_1:i}) = \lambda_{\bowtie_{S_1:i}} \sigma_{\bowtie_{S_1:i}} W \qquad \text{Single input } S_i (i > 1).$$
$$C(\zeta_1 \zeta_2) = C(\zeta_1) + T(\zeta_1) C(\zeta_2) \quad \text{Sub-sequences } \zeta_1 \text{ and } \zeta_2 \text{in join sequence } \zeta.$$
where T(*) is defined by:
$$T(\Lambda) = 1 \qquad\qquad\qquad\qquad \text{Null sequence } \Lambda.$$
$$T(S_1) = \lambda_{S_1} \qquad\qquad\qquad\quad \text{Starting input stream.}$$
$$T(\bowtie_{S_1:i}) = \sigma_{\bowtie_{S_1:i}} \lambda_{\bowtie_{S_1:i}} W \qquad \text{Single input } S_i (i > 1).$$
$$T(\zeta_1) = \prod_{k=i}^{j}(\sigma_{\bowtie_{S_1:k}} \lambda_{\bowtie_{S_1:k}} W) \;\; \text{Subsequence } \zeta_1 = (\bowtie_{S_1:i}, \ldots, \bowtie_{S_1:j-1}).$$

Each node S_q is marked by the rank, $rank(S_q) = (T(S_q) - 1)/C(S_q)$, where $C(S_q)$ and $T(S_q)$ are defined as above. This modified cost model satisfies the Adjacent Sequence Interchange (ASI) property [5] and therefore is guaranteed to produce an *optimal join order* for acyclic graphs.
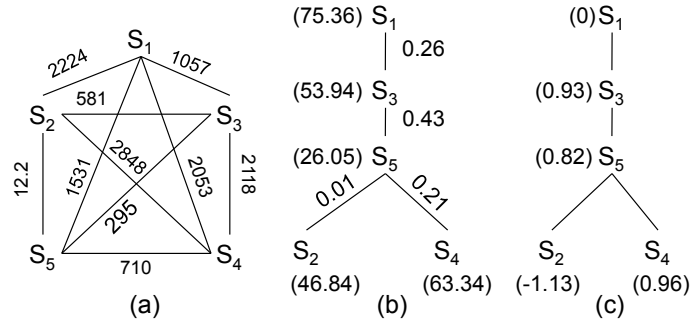

Fig. 5. a) Weighted Join Graph, b) Minimum Spanning Tree c) Rooted Tree for input S_1

Handling Cycles in TreeOpt: For the problem of optimizing cyclic join graphs the strategy is to first transform the graph into some acyclic graph and then apply TreeOpt. The aim now is to construct a good acyclic graph. Note that when ordering the pipeline of a given stream S_i, the goal is to reduce the number of intermediate tuples. Therefore, we propose to generate a *minimal spanning tree* (MST), where the weight of an edge connecting two vertexes S_i and S_j is computed as $\lambda_{S_i} * \lambda_{S_j} \sigma_{S_i S_j}$. In the static database context, [25] proposed a similar heuristic accounting for the cost of disk accesses. *TreeOpt* is then applied to produce an optimal ordering for this MST.
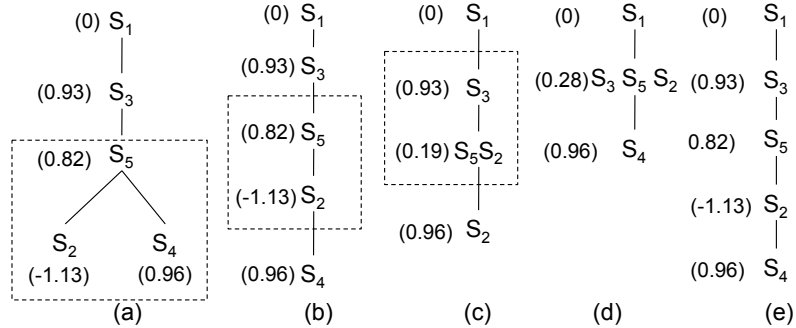

Fig. 6. Finding the Optimal Join Ordering

Example: For the weighted cyclic join graph in Figure 5.a we generate the MST shown in Figure 5.b and then compute the *rank* for each node S_q. We traverse the rooted tree (Figure 6.a) bottom up. S_5 is the first node with more than one child and we check to see if all of its children's branches are ordered by non-decreasing ranks. We then merge the children's nodes into one sequence by the ascending order of their ranks as in Figure 6.b. The resulting chain is not ordered since $rank(S_2) < rank(S_5)$ and so we merge nodes S_5 and S_2, and recompute the rank for the merged node $S_5 S_2$ (Figure 6.c). As a final step, we expand all combined nodes. The resulting sequence is the optimal join for the MST shown in Figure 5.b.
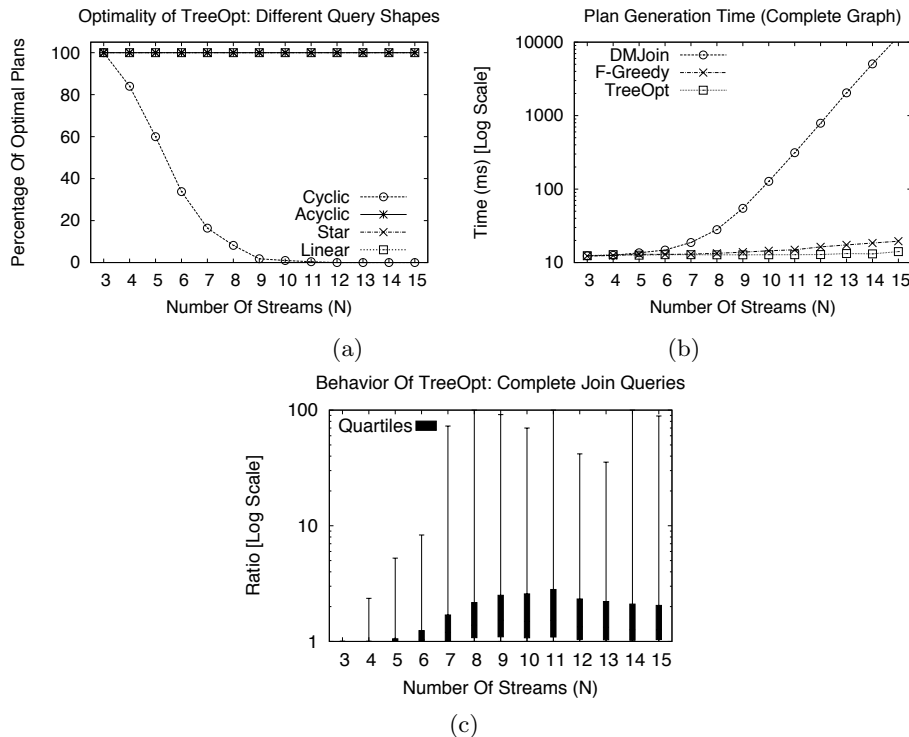

Fig. 7. a) % of Optimal Plans b) Execution Time c) Comparison: TreeOpt vs. DMJoin

Time Complexity: For a join graph with n vertexes, it takes $\mathcal{O}(n^2 log(n))$ to find a minimum spanning tree. Ordering an input stream takes $\mathcal{O}(nlog(n))$. Therefore, *TreeOpt* has a time complexity of $\mathcal{O}(n^2 log(n))$ for ordering an n-way mjoin.

Evaluation of The *TreeOpt* Algorithm. Figure 7.a depicts the percentage of runs in which *TreeOpt* generates an optimal plan. Note that all lines in Figure 7.a except for the cyclic case overlap fully. That is, *TreeOpt* generates an optimal plan for any acyclic join query as expected and has faster plan generation time than F-Greedy (as in Figure 7.b). Clearly, this is a win-win solution for acyclic join queries. However, for the cyclic queries we observe in Figure 7.b that the ability of *TreeOpt* to generate an optimal plan rapidly goes down to zero. A closer investigation of the distribution of generated plan costs reveals that in most cases the upper-quartile (top 75%) of generated plans are ≈ 2.5 fold more expensive than the optimal plan generated by the DMJoin. Additionally, we note in Figure 7.c that there are many cases when *TreeOpt* performs unacceptably, sometimes 50 fold or worse depending on the MST considered.

Summarizing: *TreeOpt* generates optimal plans for any acyclic join graph in polynomial time, while the widely used F-Greedy [15, 18, 19] often produces sub-standard plans. However, for cyclic join queries, where neither offer any guarantees on optimality, F-Greedy outperforms *TreeOpt*.

5 The FAB Algorithm

We now present our *Forward and Backward greedy* (*FAB*) algorithm. The F-Greedy algorithm incrementally picks the next best stream and is therefore too greedy at the early stages of optimization. In Figure 3, during the first iteration F-Greedy chooses S_3 as it generates the fewest number of join results. However, S_3 has a higher input rate, and its join selectivities with the remaining streams are greater than S_5. This early greedy decision affects the total plan cost by producing more intermediate tuples in comparison to the case when S_2 is chosen. By contrast, our FAB approach uses a *global impact-based ordering*, in which we explore the global impact of each stream and place the most expensive streams as the last candidate to join.

Algorithm 1 GImpactOrdering(Graph *JG*, Stream S_{start})

Input: Join Graph, *JG* of streams $\{S_1, .., S_n\}$; Start stream, S_{start}
Output: Query Plan *cPlan* for $\triangle S_{start}$
 1: **for** $i = n$ to 2 **do**
 2:      impact $= \infty$
 3:      **for** each vertex $S_q \in \{S_1, .., S_n\}$ and $S_q \neq S_{start}$ **do**
 4:          **if** GlobalImpact(*JG*, S_q) < impact **then**
 5:              impact $=$ GlobalImpact(*JG*, Node S_q); nextCandidate $= S_q$
 6:      cPlan $=$ nextCandidate $\bowtie$ cPlan; Remove nextCandidate from JG
 7: cPlan $= R_{start} \bowtie$ cPlan
 8: **return** cPlan
 9: **procedure GlobalImpact**(JoinGraph *JG*, Stream S_{curr})
10: impact $= 1$;
11: **for** each $S_x \in \{S_1, .., S_n\}$ and $S_x \neq S_{curr}$ **do**
12:      impact $=$ impact * λ_{S_x}
13:      **for** each $S_y \in \{S_1, .., S_n\}$ and $S_y \neq S_{curr}$ and $S_y \neq S_x$ **do**
14:          impact $=$ impact * $\sigma_{S_x S_y}$
15: return impact

The *global impact* metric used by FAB is listed in Algorithm 1 (Line 12). The global impact of stream S_q ($S_q \in S$ and $S_q \neq S_{start}$) is the product of the arrival rates of all remaining streams and the join selectivities not related to S_q. The intuition here is that if a stream, S_q, has the least impact, i.e., high input rate and poor selectivities, then the generated plan will be the most expensive. Following this principle, FAB starts by picking the candidate that has the least impact and places it as $\bowtie_{S_{start}:n-1}$. Next, we remove this candidate stream from the join graph and proceed to determine the next-to-last stream to join in the update pipeline of stream S_{start}. This process is done iteratively until all $n-1$ positions in the pipeline are filled. Our FAB approach makes use of the *global impact ordering* in unison with the F-Greedy to generate optimal plans for ordering multi-join queries.

Time complexity: The global impact ordering has a complexity of $\mathcal{O}(n^2)$. Therefore, the time complexity to order the pipeline for processing new tuples from a single stream by FAB is $\mathcal{O}(n^2)$; thus, ordering an *n-way* mjoin operator has complexity $\mathcal{O}(n^3)$.
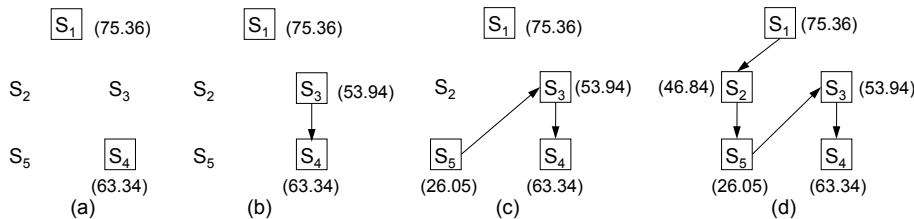
Fig. 8. Finding Optimal Join Ordering Through GImpactOrdering (as described in Algorithm 1)

Example: Consider $Q1$ and the processing of tuples from stream S_1. We compute the global impact of all remaining join participants. As in Figure 8, the candidate with the least impact is identified, e.g., the impact of S_4 is ≈ 300 while that of S_3 is $\approx 11K$ and is therefore the last stream to be joined in S_1's pipeline i.e., $\bowtie_{S_{1:4}} = S_4$. Iteratively, the resultant ordering is $S_1 \bowtie S_5 \bowtie S_2 \bowtie S_3 \bowtie S_4$ and generates ≈ 3629 intermediate tuples/sec. This is equivalent to the optimal plan generated by DMJoin. Recall in Section 3.2, F-Greedy returns a sub-standard plan that generates ≈ 7212 intermediate tuples/sec, which is **2** fold more expensive.

Evaluation of the FAB Algorithm. In Figure 9.b, the FAB approach has a higher likelihood of generating an optimal for the entire spectrum of queries than F-Greedy (Figure 9.b). However, FAB does not provide optimality guarantees as TreeOpt (Figure 7.b) for acyclic queries.
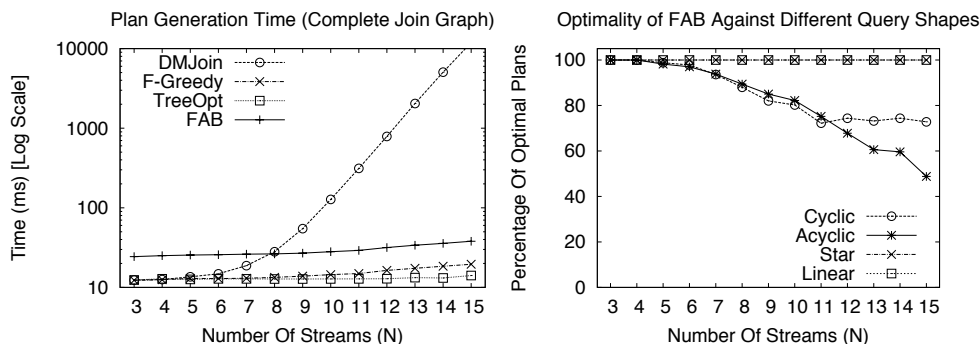

Fig. 9. FAB vs. DMJoin: (a) Plan Generation Time (b) % of Optimal Plans Generated by FAB

For acyclic join queries, FAB produces near-optimal plans with the upper quartile of the runs generating optimal plans (as in Figure 10.a). FAB is shown to generate plans that are at-most 1.25 fold more expensive in cost than those generated by DMJoin. Figure 10.b highlights similar trends in the upper quartile of runs when processing cyclic join queries. The most expensive plan in FAB are cyclic queries, which are at most 2 fold more expensive than those generated by DMJoin.

6 The Q-Aware Approach

Due to the NP-completeness of the query optimization problem no single algorithm can effectively and efficiently handle the entire spectrum of join queries. We therefore present our *query shape aware approach* (**Q-Aware**) that is sensitive to query shape. This approach has two steps. First we determine the shape of the join graph; next, based on the query shape, we choose the algorithm to generate an optimal or a good plan, as described in Table 6. An acyclic query is always ordered using TreeOpt as it is guaranteed to return an optimal plan. For cyclic queries, the approach uses the FAB technique to generate a good plan.
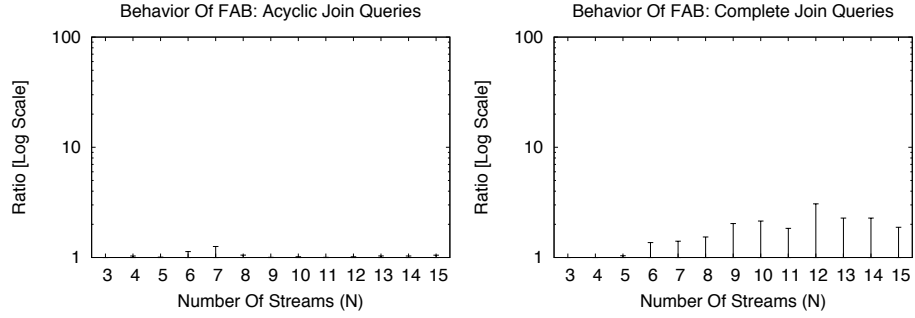
Fig. 10. FAB vs. DMJoin (Plan Cost Comparison): (a) Acyclic (b) Complete

Table 2. Summarizing *Q-Aware* Approach

Query (Join Graph) Shape	Algorithm	Complexity	Properties
Linear			
Star	TreeOpt	$\mathcal{O}(n^2 log(n))$	Optimal
General-Acyclic			
Cyclic	FAB	$O(n^3)$	Near-optimal

Time complexity: For a join graph with n vertexes (streams) and e edges (selectivities) the time complexity to determine if a given join query has cycles is $\mathcal{O}(n+e)$. Since the number of edges is smaller than $\mathcal{O}(n^3)$, the time complexity of *Q-Aware* $\mathcal{O}(n^3)$ for ordering an n-way mjoin.

7 Conclusion

Streaming environments require their query optimizers to support (1) re-optimization and migration techniques to keep up with fluctuating statistics, (2) have polynomial time complexity, (3) increase the probability of generating optimal plans, and in the worst case scenarios it must aim to generate a good plan, and (4) handle a diversity of query types. Motivated by this, we revisit the problem of continuous query optimization. We begin by experimentally studying the effectiveness of the state-of-the-art continuous query optimization techniques for different query shapes which confirms that these techniques generate sub-standard plans even for the simple problem of ordering acyclic queries. To tackle this deficiency, we extend the classical *IK algorithm* to the streaming context called *TreeOpt*. TreeOpt is a polynomial-time algorithm, and is superior to the greedy approach for general acyclic queries. For the harder problem of ordering cyclic graphs, *TreeOpt* is not be a viable alternative. Therefore, we introduce our *FAB* algorithm utilizes our *global impact ordering* technique. This approach increases the chances of finding an optimal plan as well as generating a less expensive plan when an optimal plan cannot be found in polynomial time. Lastly, we put forth our *Q-Aware* approach that generates optimal or near-optimal plans for any query shape in guaranteed polynomial time.

Acknowledgement

References

1. 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 Bernstein, P.A., ed.: SIGMOD. (1979) 23–34

2. Vance, B., Maier, D.: Rapid bushy join-order optimization with cartesian products. In: SIGMOD. (1996) 35–46
3. Kossmann, D., Stocker, K.: Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst. **25**(1) (2000) 43–82
4. Moerkotte, G., Neumann, T.: Dynamic programming strikes back. In: SIGMOD. (2008) 539–552
5. Ibaraki, T., Kameda, T.: On the optimal nesting order for computing n-relational joins. ACM Trans. Database Syst. **9**(3) (1984) 482–502
6. Swami, A.N., Iyer, B.R.: A polynomial time algorithm for optimizing join queries. In: ICDE. (1993) 345–354
7. Ioannidis, Y.E., Kang, Y.C.: Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In: SIGMOD. (1991) 168–177
8. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR. (2005) 277–289
9. Ali, M.H., Aref, W.G., Bose, R., Elmagarmid, A.K., Helal, A., Kamel, I., Mokbel, M.F.: Nile-pdt: A phenomenon detection and tracking framework for data stream management systems. In: VLDB. (2005) 1295–1298
10. Rundensteiner, E.A., Ding, L., Sutherland, T.M., Zhu, Y., Pielech, B., Mehta, N.: Cape: Continuous query engine with heterogeneous-grained adaptivity. In: VLDB. (2004) 1353–1356
11. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: SIGMOD. (2002) 49–60
12. Ayad, A., Naughton, J.F.: Static optimization of conjunctive queries with sliding windows over infinite streams. In: SIGMOD. (2004) 419–430
13. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: Stream: The stanford stream data manager. In: SIGMOD Conference. (2003) 665
14. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: CIDR. (2003)
15. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: SIGMOD. (2004) 407–418
16. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic plan migration for continuous queries over data streams. In: SIGMOD. (2004) 431–442
17. Bukowski, R., Peacock, R., Averill, J., Cleary, T., Bryner, N., Walton, W., Reneke, P., Kuligowski, E.: Performance of Home Smoke Alarms: Analysis of the Response of Several Available Technologies in Residential Fire Settings. NIST Technical Note 1455 (2000) 396
18. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB. (2003) 285–296
19. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: VLDB. (2003) 500–511
20. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In: ICDE. (2003) 341–352
21. Hammad, M.A., Franklin, M.J., Aref, W.G., Elmagarmid, A.K.: Scheduling for shared window joins over data streams. In: VLDB. (2003) 297–308
22. Ganguly, S., Hasan, W., Krishnamurthy, R.: Query optimization for parallel execution. In: SIGMOD. (1992) 9–18
23. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: SIGMOD. (2006) 431–442
24. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: Rpj: Producing fast join results on streams through rate-based optimization. In: SIGMOD. (2005) 371–382
25. Krishnamurthy, R., Boral, H., Zaniolo, C.: Optimization of nonrecursive queries. In Chu, W.W., Gardarin, G., Ohsuga, S., Kambayashi, Y., eds.: VLDB. (1986) 128–137