Worcester Polytechnic Institute
DigitalCommons@WPI

Computer Science Faculty Publications | Department of Computer Science

11-1-2009

A New Look At Generating Multi-Join Continuous Query Plans: A Qualified Plan Generation Problem

Yali Zhu
Oracle Corporation, yali.zhu@oracle.com

Venkatesh Raghavan
Worcester Polytechnic Institute, venky@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

Follow this and additional works at: http://digitalcommons.wpi.edu/computerscience-pubs

Part of the Computer Sciences Commons

Suggested Citation

Zhu, Yali , Raghavan, Venkatesh , Rundensteiner, Elke A. (2009). A New Look At Generating Multi-Join Continuous Query Plans: A Qualified Plan Generation Problem. .
Retrieved from: http://digitalcommons.wpi.edu/computerscience-pubs/20

This Other is brought to you for free and open access by the Department of Computer Science at DigitalCommons@WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of DigitalCommons@WPI.

WPI-CS-TR-09-13 Nov 2009

A New Look At Generating Multi-Join Continuous Query Plans: A Qualified Plan Generation Problem

by

Yali Zhu
Venkatesh Raghavan
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

A New Look At Generating Multi-Join Continuous Query Plans: A Qualified Plan Generation Problem

Yali Zhu[a], Venkatesh Raghavan[b], Elke A. Rundensteiner[b]

[a] *Oracle Corporation, Redwood Shores, CA USA*
[b] *Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA USA*

Abstract

State-of-the-art relational and continuous algorithms alike have focused on producing optimal or near-optimal query plans by minimizing a single cost function. However, ensuring accurate yet real-time responses for stream processing applications necessitates that the system identifies qualified rather than optimal query plans - with the former guaranteeing that their utilization of both the CPU and the memory resources stays within their respective system capacities. In such scenarios, being optimal in one resource usage while out-of-bound in the other is not viable. Our experimental study illustrates that to be effective a qualified plan optimizer must explore an extended plan search space called the jtree space composed not only of the standard mjoin and binary join plans, but also of general join trees with mixed operator types. While our proposed dynamic programming-based JTree-Finder algorithm is guaranteed to generate a qualified query plan if such a plan exists in the search space, its exponential time complexity makes it not viable for continuous stream environments. To facilitate run-time optimization, we thus propose an efficient yet effective two-layer plan generation framework. The proposed framework first exploits the positive correlation between the CPU and memory usages to obtain plans that are minimal in at least one of the two resource usages. In our second layer we propose two alternative polynomial-time algorithms to explore the negative correlation between the resource usages to successfully generate query plans that adhere to both CPU and memory resource constraints. Effectiveness and efficiency of the proposed algorithms are experimentally evaluated by comparing them to each other as well as state-of-the-art techniques.

Key words: multi-constraint query optimization, continuous queries, streaming

Email addresses: yali.zhu@oracle.com (Yali Zhu), venky@cs.wpi.edu (Venkatesh Raghavan), rundenst@cs.wpi.edu (Elke A. Rundensteiner)

1. Introduction

1.1. Continuous Query Plan Generation

State-of-the-art query optimization algorithms in static databases [1, 2, 3, 4, 5] primarily focus on generating an optimal or near-optimal plan by minimizing a single cost function, typically the total processing costs comprised of I/O or CPU [6]. Continuous query processing [7, 8] differs from its static counterpart in several aspects. First, the incoming streaming data is unbounded and the query lifespan is potentially infinite. Therefore, runtime output rate is a better metric than the total CPU time needed to handle all input data [9]. When the per-unit-time CPU usage of a query plan is less than the available system CPU capacity, the query execution is able to keep up with incoming tuples and produce real-time results at an optimal output rate [10].

Second, real-time response requirements make continuous queries memory resident [7]. Stateful operators, such as joins, store input tuples in states with which future incoming tuples of other streams will join. In time-critical applications, such as fire-sensor monitoring, it is common to have multi-join queries with large numbers of participant streams with high input rates. In such scenarios, the size of the in-memory operator states could potentially grow to be very large, making memory a precious resource. Memory overflow can result in unacceptable outcomes, such as temporary halt of query execution [9, 11, 12], approximation of query results [13] and in some cases thrashing.

To summarize, generating a query plan that is optimal in one resource usage while out-of-bound in the other is not an acceptable solution. Therefore, the aim is to generate a query plan with both resource consumptions within their respective system resource capacities, henceforth called a *qualified plan* [10]. All qualified plans are guaranteed to produce results at the same output rate [10].

To address this qualified plan generation problem, one could attempt to design a combined (singular) cost function that captures both resource usages. This would be beneficial as we could then capitalize on state-of-the-art optimization techniques. However, such an approach suffers from drawbacks that make it unsuitable. First, a singular cost function that captures both CPU and memory usages and their correlation *a priori* is in practice hard to obtain [14]. This is because the problem is no longer a minimization problem but rather a system resource constraint satisfaction problem. Also, there is no monotonic clearly characterizable relationship between the resources. On the contrary, we show that these resources in parts of the search space may be positively correlated and in others negatively correlated. Second, a query plan that is minimal by this new singular function need not be optimal or near-optimal in either resource usage nor guaranteed to be qualified. Additionally we note that the problem is NP-hard [15], yet efficient algorithms are a must in the streaming context for runtime optimization. Thus, we now set out in this work to provide a fresh approach to this qualified plan generation problem.

1.2. Relationship Between Resource Usages

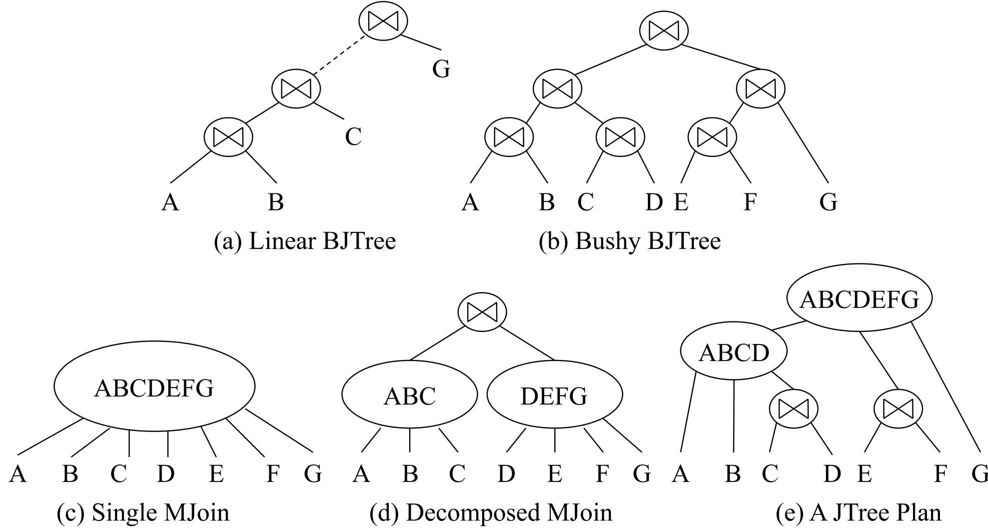


Figure 1: JTree Solution Space: (a) & (b) Traditional BJTrees, (c) MJoin [9] (d) De-composed MJoin [9], (e) A JTree Plan Not Considered In Literature

Similar to other multi-objective optimizations [16, 17, 14], we first characterize the relationship among the determining cost factors. The observation that a query plan with less data (less memory) typically requires less CPU processing time is well known. This direct correlation is referred to in this work as a **positive correlation**. State-of-the-art optimizers in static databases [1, 2, 3, 5, 18] as well as streaming databases [9, 15, 19] exploit this positive correlation by minimizing intermediate results (memory) with the assumption that this will also reduce CPU costs.

To illustrate this, consider the two commonly used methods for executing continuous joins: binary join trees (*bjtree*) [20] and multi-way join operators (*mjoin*) [19, 9, 15, 10]. A *bjtree* is composed of binary join operators that store intermediate results, while an *mjoin* is a single operator that takes as input all participant streams. The new tuples from each stream in mjoin are joined with the remaining streams in a particular order. Existing optimization techniques for both these join methods aim to minimize the total number of intermediate results [15]. In bjtree, this reduces the memory required to store intermediate results as well as the CPU costs for future joins. On the other hand, in mjoin, this reduces the CPU costs needed to recompute intermediate tuples.

However, **negative correlation** between CPU and memory usage could also arise for multi-join query plan optimization. In other words, an increase in the usage of one resource may decrease the usage of the other. While the concept of *negative correlation*

is well known, to the best of our knowledge none of the state-of-the-art approaches exploit both positive and negative correlations to generate a query plan that is both CPU and memory resource adherent. More precisely, rather than only considering different bjtrees by exploiting only positive correlation, our approach explores removing some intermediate states to reduce memory usage at the expense of increasing the re-computation CPU costs, thus also exploiting the negative correlation between resources. Conversely, rather than choosing between different mjoin orderings, some input streams can be combined to form binary joins whose results are stored in intermediate states and fed into the mjoin operator. This reduces CPU costs incurred by the re-computation at the expense of increasing memory usage. In Section, 2.4 we present a formal analysis highlighting these resource trade-offs.

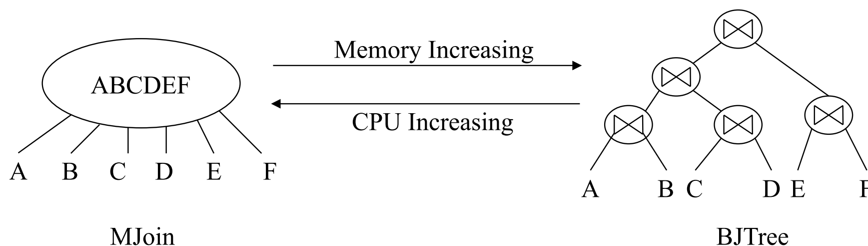


Figure 2: Migrating Between MJoin and BJTree

State-of-the-art algorithms in multi-join query optimization stay within one join method by exploiting only the positive correlation in each join method. That is, they explore either solely within the bjtree [5, 2, 10] or the mjoin [19, 15] solution space. [9], which were the first to introduce the idea of mjoin, also observed that there is a limit in the number of input streams for an mjoin to be effective. They considered decomposing a single mjoin operator into two smaller mjoin operators as in Figure 1.d. [9] does not consider memory resource utilization (assuming it to be always sufficient) and follows the traditional approach of optimizing the query by simply reducing CPU processing time. Thus [9] fails to explore a general join plan, as in Figure 1 e, composed of a mixture of both mjoin and binary join operators at any level henceforth referred to as **jtrees**. In our experimental analysis, we observe that these existing algorithms thus may not generate qualified query plans even when a qualified plan exists in the solution space.

1.3. The Proposed Approach

We first present our dynamic programming-based **JTree-Finder** algorithm that explores the complete jtree search space and guarantees the generation of a qualified plan. This guarantee comes at a high complexity cost, making it not practical for runtime optimization needed for streaming applications.

We thus design a polynomial-time qualified plan generation solution in the form of a **two-layer framework**. In the first layer, the 2-dimensional problem is transformed into a 1-dimensional minimization problem. This allows us to employ state-of-the-art solutions [15, 19, 1, 5, 10] that exploit positive correlation among the two resources to generate a good mjoin or a good bjtree plan respectively. This layer either returns 1) a qualified mjoin or bjtree plan, if found, or 2) a negative result to denote that the available resources are too limiting and hence no qualified plan exists, or 3) triggers the second layer of plan generation into action.

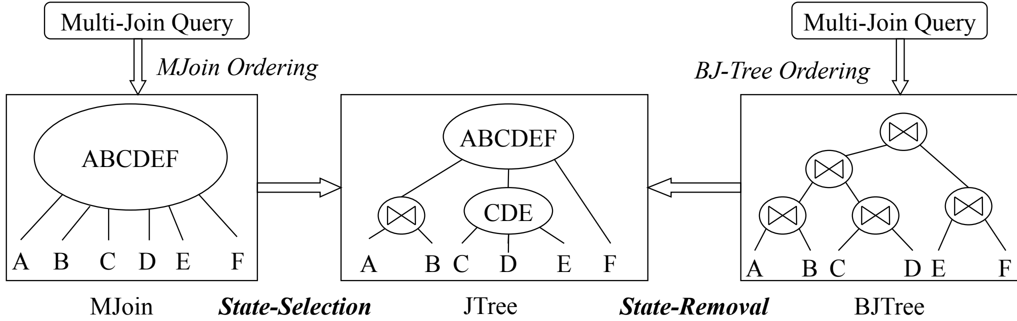


Figure 3: Two Strategies In Optimization

The second layer of optimization piggybacks on the first layer by using the generated best possible mjoin or bjtree plans as the starting point as they provide lower bounds on memory or CPU utilization respectively. For the second layer we propose two polynomial-time **hill-climbing** search algorithms, named *state-selection* and *state-removal*. Both exploit the negative correlation between CPU and memory usage to generate a qualified jtree. More precisely, *state-selection* starts with the previously generated mjoin-based plan (i.e., guaranteed to be minimal in memory usage) and reduces the excessive CPU resource utilization by sacrificing some memory resources (Figure 3). In this process, we aim to save on CPU costs wasted on the re-computations of intermediate results. On the other hand, *state-removal* starts with a good bjtree-based plan (i.e., optimal or near-optimal in CPU usage), and aims to reduce the excessive memory usage by selecting intermediate states to be removed at the expense of increasing CPU resources needed for their re-computation.

1.4. Summary of Contributions

- We position the generation of multi-join continuous query plans as a qualified plan generation problem with the aim of generating a query plan whose usages of both CPU and memory resources are within the system capacities.
- We consider the extended search space of qualified plans that incorporates general jtree-based plans. We employ the *JTree-Finder* algorithm for exploring this search

space. Our experimental evaluation demonstrates the need to indeed consider this extended plan space. It further highlights the need for a polynomial-time approach. In this performance study, this dynamic progamming technique is not feasible for stream processing systems due to its exponential time complexity.

- We propose an alternative polynomial-time two-layer plan generation framework, which exploits existing technologies to first generate an mjoin and a bjtree plan that are minimal in memory and CPU resources respectively, exploiting the positive correlation between the resources usages.

- We present two new hill-climbing algorithms, namely *state-selection* and *state-removal* that exploit the negative trade-offs between CPU and memory usages. The former uses the mjoin-plan and the latter uses the bjtree-plan, both generated in our first layer of optimization, as their respective starting point.

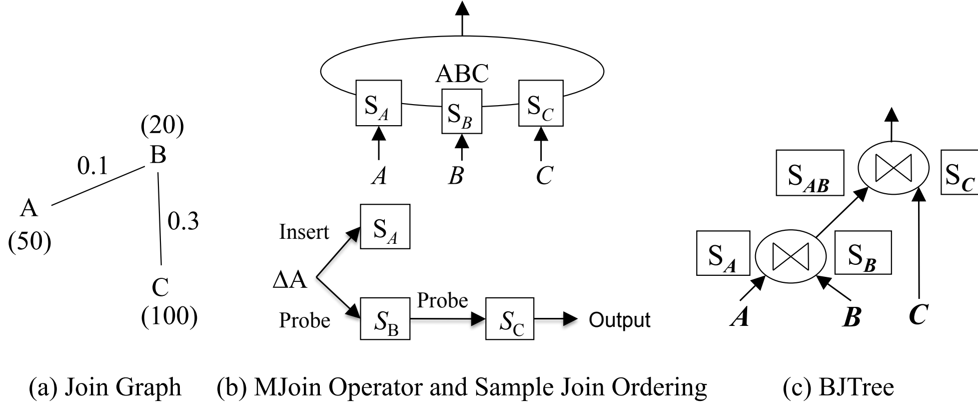
- We show the effectiveness and efficiency of our algorithms through our second set of experiments that compares them against each other and to popular approaches. Our proposed *two-layer optimization* is shown to as effective in a large variety of testing scenarios as the exhaustive approach in finding a qualified plan. Additionally, due its polynomial-time execution it is able to generate qualified plan even when for a large number of streams, when the dynamic programing solution is to expensive.

1.5. Organization

In Section 7 we survey the related work. We analyze the CPU and memory cost models to reveal the conditions under which these resources have a positive versus a negative correlation in Section 2. In Section 3, we present various resource settings that would require the search space to be extended to include general jtrees. Additionally, here we also present our dynamic programming based *JTree-Finder* algorithm. Section 4 experimentally highlights the high time complexity of JTree-Finder. We propose our polynomial-time two-layer plan generation framework in Section 5. The effectiveness of our approach is presented in Section 6 through our experiments. Section 8 concludes the paper.

2. Preliminaries

In this section, we analyze the CPU and memory costs for the state-of-the-art methods of implementing multi-join continuous queries, namely mjoin and bjtree. This comprehensive analysis reveals the conditions under which CPU and memory usage have a positive or a negative correlation.



(a) Join Graph (b) MJoin Operator and Sample Join Ordering (c) BJTree

Figure 4: Commonly Used Join Methodologies

2.1. Join Method Basics

The *join graph* (JG) (in Figure 4.a) represents a multi-join query along with statistical information such as input rates, selectivities, etc. A vertex in the join graph represents an input stream, marked by its stream name and arrival rate. An edge between two vertices indicates a join predicate between the two streams and is marked by the join selectivity. For simplicity, henceforth we assume independent join selectivities. In principle a richer selectivity estimation-model [21, 22] could also be utilized. However, this would further complicate the already NP-Hard problem considered in this work.

Figures 4.b and c show *mjoin* and *bjtree* respectively, two common practices used for executing multi-join queries over windowed streams. [9] proposed a multi-way join operator called mjoin to process a multi-join query over windowed streams. An mjoin operator is a single multi-way operator that takes as input the continuous streams from all join participants. 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 no longer stored, saving space. To illustrate, in Figure 4.b new tuples from A (ΔA for short) are first inserted into the state of A (denoted as S_A), then used to probe the state of B (S_B), and the resulting join tuples then go on to probe the state of stream C (S_C) to produce the final output result. The key idea is that the order in which the new tuples from stream A are processed (joined) is independent from the order in which tuples from streams B and C are processed. Alternative t mjoin a traditional plan (as shown in Figure 4.c) containing binary join operations can also be used to process continuous multi-join queries.

The algorithms proposed in this work are general and not restricted by any particular physical implementation of mjoin or bjtree like hash vs. nested-loop or particular window type like count- vs. time-based. To anchor our analysis, our cost analysis below makes use

Term	Meaning
C_i	Cost of inserting a tuple to a state (ms)
C_d	Cost of deleting a tuple from a state (ms)
C_j	Cost of joining a pair of tuples (ms)
λ_X	Average input rate from stream X (tuples/sec)
σ_{XY}	Selectivity of join $X \bowtie Y$
W	Sliding time-based window constraint measured in seconds
$\lvert S_X \rvert$	Number of tuples in state S_X

Table 1: Terms Used in Cost Models

of the widely adopted symmetric stream hash join [9] with time-based windows [23, 8]. Prior work has shown that for most cases of a hash based join implementation is superior to other implementations such as tree-based and nested-loop join [10]. Each join operator keeps one *state* per input stream, storing tuples in one window frame for future joins. For ease of discussion, we assume all join predicates to have the same window size, though the techniques proposed in this work are not restricted by it.

We apply the commonly adopted per-unit-time cost metric [23], in which the CPU cost is the CPU processing time required to process all tuples arriving in one time unit. Table 1 explains the terms used in our cost-model. The cost function CPU($\mathcal{P}$, JG) returns the CPU utilization and Memory($\mathcal{P}$, JG) returns the memory resources needed to execute the query plan $\mathcal{P}$ realizing the user query caaptured by the join graph JG.

2.2. *Cost Analysis for MJoin*

CPU costs for the mjoin in Figure 4(b) is the cumulative cost of processing tuples from streams A, B and C. Based on the optimal join orderings in Figure 4(b), a new tuple from A is first inserted into state S_A (at cost C_i). Existing tuples that are now outside the window frame are purged[1] from S_A (at cost C_d). This inserted tuple is then joined with tuples in state S_B and the resulting tuples are used to join with tuples in state S_C. A similar process applies to tuples from B and C. The CPU costs for input A in a unit time are $CPU_A = \lambda_A(C_i + C_d) + \lambda_A|S_B|\sigma_{AB}C_j + \lambda_A|S_B||S_C|\sigma_{ABC}C_j$, where $\sigma_{ABC} = \sigma_{AB}\sigma_{BC}$, $|S_B| = \lambda_B W$, and $|S_C| = \lambda_C W$.

$$CPU_A = \lambda_A(C_i + C_d) + \lambda_A\lambda_B\sigma_{AB}WC_j + \lambda_A\lambda_B\sigma_{AB}\lambda_C\sigma_{BC}W^2C_j \qquad (1)$$

[1] We assume self-purge, while cross-purge is also applicable.

The total CPU processing costs for our mjoin are:

$$\begin{aligned} CPU_{mjoin} &= CPU_A + CPU_B + CPU_C \\ &= (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) + \lambda_B \lambda_C \sigma_{BC} W C_j \\ &+ 3\lambda_A \lambda_B \lambda_C \sigma_{ABC} W^2 C_j + 2\lambda_A \lambda_B \sigma_{AB} W C_j \end{aligned} \tag{2}$$

The memory cost of an mjoin, immaterial of the chosen join ordering, is the same and fixed to be the total state size for maintaining input stream tuples. This cost is relatively stable through out the life span of the query. The run-time memory costs may fluctuate as intermediate tuples temporarily exist and can be minimized by choosing the optimal join orderings. The memory costs for the mjoin (Figure 4(b)) thus is estimated as:

$$MEM_{mjoin} = |S_A| + |S_B| + |S_C| = \lambda_A W + \lambda_B W + \lambda_C W \tag{3}$$

2.3. Cost Models for BJTree

Similarly, the CPU costs of a bjtree are the cumulative cost of processing tuples from each input stream in one time unit. In Figure 4, a new tuple from A is first inserted into the state S_A and old tuples from S_A are deleted. The new tuple then joins with tuples in state S_B. The joined tuples are inserted into intermediate state S_{AB} and older tuples in S_{AB} are deleted. These joined tuples finally join with tuples in state S_C. Tuples from input B follow similar steps while tuples from input C directly join with tuples in state S_{AB}. The cost models to compute the unit CPU costs for input A is $CPU_A = \lambda_A(C_i + C_d) + \lambda_A|S_B|\sigma_{AB}(C_j + C_i + C_d) + \lambda_A|S_B||S_C|\sigma_{ABC}C_j$. The CPU costs for input B are identical to A. However the CPU costs of stream C is $CPU_C = \lambda_C(C_i + C_d) + \lambda_C|S_{AB}|\sigma_{BC}C_j$. Given $|S_{AB}| = \lambda_A\lambda_B\sigma_{AB}W^2$ and $|S_B| = \lambda_B W$ we have,

$$\begin{aligned} CPU_{bjtree} &= (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) + 3\lambda_A \lambda_B \lambda_C \sigma_{ABC} W^2 C_j \\ &+ 2\lambda_A \lambda_B \sigma_{AB} W (C_j + C_i + C_d) \end{aligned} \tag{4}$$

Estimated memory costs is given by the total state size:

$$MEM_{bjtree} = |S_A| + |S_B| + |S_C| + |S_{AB}| = MEM_{mjoin} + \lambda_A \lambda_B \sigma_{AB} W^2 \tag{5}$$

The first two terms in Equation 4 are identical for bjtrees of any shape. The third term, $2|S_{AB}|(C_j + C_i + C_d)$, is join-order-dependent. Choosing a better join ordering lowers the size of intermediate states, which decreases the memory cost as indicated by Equation 5, and also lowers the CPU costs as indicated by Equation 4. Hence in bjtree, CPU and memory costs are positively correlated.

2.4. Condition For Negative Correlation

As indicated by Equations 3 and 5, MEM_{bjtree} is always larger than MEM_{mjoin} as the bjtree stores all intermediate states. So a negative correlation between CPU and memory may exist when the CPU costs of bjtree are smaller than the CPU costs of mjoin. At first glance, this seems to always hold, because without storing intermediate results, mjoin requires extra CPU resources for re-computation. However, bjtree also needs extra CPU resources to maintain intermediate states. The recomputation of the state BC requires $CPU_{mjoin_recomput}$ while $CPU_{bjtree_maintain}$ is the cost to maintain the state AB. From Equations 2 and 4, we have:

$$CPU_{mjoin_recomput} = \lambda_B \lambda_C \sigma_{BC} W C_j \tag{6}$$

$$CPU_{bjtree_maintain} = 2\lambda_A \lambda_B \sigma_{AB} W (C_i + C_d) \tag{7}$$

From this, several key observations can be drawn,

1. When $CPU_{mjoin_recomput} > CPU_{bjtree_maintain}$, a bjtree by storing intermediate results uses more memory but is able to reduce the CPU usage as no re-computations are needed.
2. When $CPU_{mjoin_recomput} < CPU_{bjtree_maintain}$, an mjoin by having extra re-computation CPU costs saves valuable memory resources needed to store them as well as the CPU resources needed to maintain them.

In this work, we exploit this phenomenon of sacrificing one resource to gain another.

3. Qualified Plan Solution Space

3.1. The Basics on Optimal vs. Qualified Plans

Definition 1. *A query plan $\mathcal{P}$ for the join graph JG is an* **optimal plan** *with respect to CPU (or memory) usage, if there does not exist a plan $\mathcal{P}' \neq \mathcal{P}$ for JG such that CPU($\mathcal{P}'$, JG) < CPU($\mathcal{P}$, JG) (or Memory($\mathcal{P}'$, JG) < Memory($\mathcal{P}$, JG)).*

State-of-the-art algorithms in traditional [1, 2, 3, 4, 5] and continuous query systems [15, 9, 19] focus on generating a query plan that is minimal in one cost function (such as in CPU or in memory usage). They however do not tackle the problem addressed in this work of generating qualified plans that adhere to both CPU and memory usage constraints. To elaborate, assume the query $\mathcal{Q}$ represented by a join graph JG and the available system resources CPU_{avail} and Mem_{avail}. Let $\mathcal{P}_1$ be the optimal and $\mathcal{P}_2$ be a near-optimal plan for JG in regards to CPU resources. Let us also assume that both plans meet the system CPU threshold, i.e., CPU($\mathcal{P}_1$,JG) $<$ CPU($\mathcal{P}_2$,JG) $< CPU_{avail}$. It

is easy to envision a scenario where the near-optimal plan $\mathcal{P}_2$ meets the memory threshold while the optimal plan $\mathcal{P}_1$ exceeds the system memory resources, i.e., Memory($\mathcal{P}_2$,JG) $< Mem_{avail} <$ Memory($\mathcal{P}_1$,JG). In such scenarios the state-of-the-art algorithms that minimize CPU costs will produce query plan $\mathcal{P}_1$ that does not adhere to both the resource constraints, while missing the qualified plan $\mathcal{P}_2$.

Definition 2. *Given the available system resources CPU_{avail} and Mem_{avail}, a plan $\mathcal{P}$ for JG is a* **qualified plan** *if CPU($\mathcal{P}, JG$) $\leq CPU_{avail}$ and Memory($\mathcal{P}, JG$) $\leq Mem_{avail}$.*

The **objective of this work** is to design algorithm(s) that generate a qualified plan as in Definition 2.

3.2. The JTree Solution Space

To better understand the entire solution space, we investigate all possible scenarios depicted in Figure 5 where we vary the system capacities CPU_{avail} and Mem_{avail}, and the CPU and memory utilization of popular join methods bjtree and mjoin. In Figures 5.a, 5.b and 5.c the estimated CPU and memory usages of bjtree and/or mjoin lie below their respective system constraints. In such situations the existing optimization techniques can be successfully applied. That is, in the case of Figure 5.a we can pick either the mjoin or the bjtree, as both are qualified. However, in Figure 5.b only the bjtree and in Figure 5.c only the mjoin solution is a qualified solution and therefore we choose accordingly.

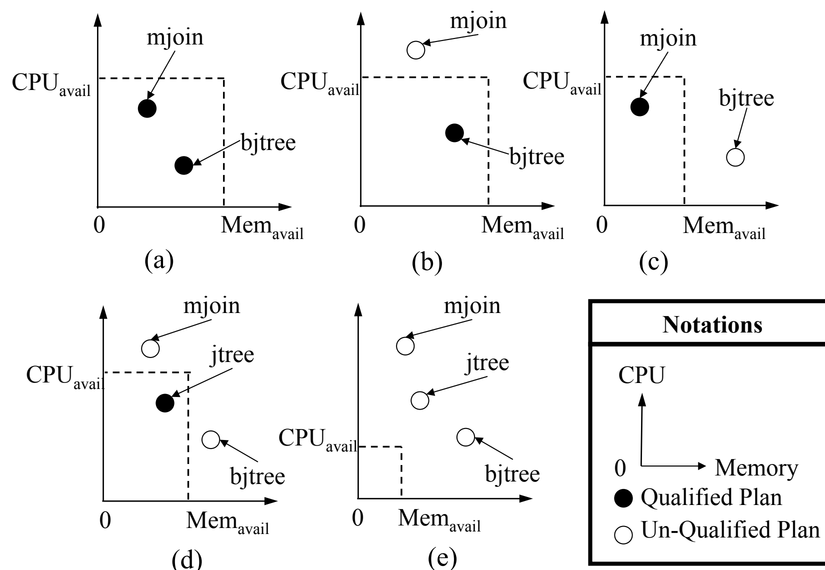


Figure 5: Various System Resource Settings

Now consider the scenario in Figure 5.d where the CPU utilization of mjoin and the memory utilization of the bjtree are above their respective available system resources. We observe that in such scenarios where neither mjoin nor bjtree query plans are qualified, a qualified plan may nonetheless exist. This is achieved by exploiting the *negative correlation* that arises between CPU and memory utilization (as discussed in Section 2.4). To elaborate, in Figure 5.d the CPU utilization of mjoin (CPU_{mjoin}) is above the system CPU threshold. While the memory utilization of mjoin (Mem_{mjoin}) is well below the available system memory resources. In such scenarios we can sacrifice memory resources by storing some intermediate results and therefore saving CPU resources that would have otherwise been wasted for their re-computation. The resulting query plan is a *jtree* whose memory and CPU resource consumptions meet their respective resource constraints (as in Figure 5.d). As neither the pure mjoin operator nor a pure bjtree are viable at all times, we now introduce a generalization of a join plan that enables us to profit from this negative resource correlation.

Definition 3. *A* **join tree (JTree)** *for a query Q represented by a join graph JG is a query plan where each node is a join with arity, $k \geq 2$. $k = 2$ implies that it is a symmetric binary join, while $k \geq 3$ means that the node is an mjoin operator.*

The intuition is to have a join solution that now exploits the benefits of both mjoin and bjtree join methods. Our experimental evaluation supports the claim that considering jtrees in the solution space increases the possibility of finding qualified plans (See Section 4). Lastly, scenarios (as depicted in Figure 5.e) where the available system resources are too restrictive, no qualified plan exists. In such cases, a deployed system would have to resort to applying more drastic approaches such as load-shedding [13] and memory-spilling [11, 12] to reduce the load of the system. These techniques incur loss of result accuracy or delays due to the addition of I/O costs. Since these strategies are orthogonal to our topic of generation of query plans that are CPU and memory resource adherent, we refer the reader to the literature to learn more about these methodologies [11, 13, 12].

In summary, state-of-the-art techniques avoid including general jtree shaped plan as this makes the search space considerably bigger and therefore have longer execution time. However, generating an optimal plan for one resource usage while being out-of-bound in another resource is not a viable solution in our context. Figure 5 clearly demonstrates that exploring the extended search is indeed useful and can potentially avoid the repeated triggering re-optimization or load shedding.

3.3. Dynamic Programming-Based Exploration Of JTree Space

To understand the search space and provide the baseline to compare our polynomial-time heuristics, we now extend the classical bottom-up dynamic programming approach

[2] to now search the extended jtree solution space and generate qualified plans. Henceforth, it is called *JTree-Finder*. For a given user query and system resource constraints, if a qualified jtree plan exists in the solution space, JTree-Finder is guaranteed to find it. In this work, we primarily introduce JTree-Finder and its results as a benchmark for evaluating the effectiveness our heuristic-based two-layer qualified plan generation framework presented in Section 5. In principle, other dynamic programming techniques [24, 5] and pruning techniques [25] could similarly be employed.

JTree-Finder differs from the classical algorithm [2, 5, 24] in several aspects. First, rather than just looking into left-deep bjtrees [2], the proposed algorithm explores **the much richer jtree solution space**. The complete search space contains all possible query plan shapes, including *bushy bjtrees*, *mjoins* and *jtrees*, as in Figure 1.

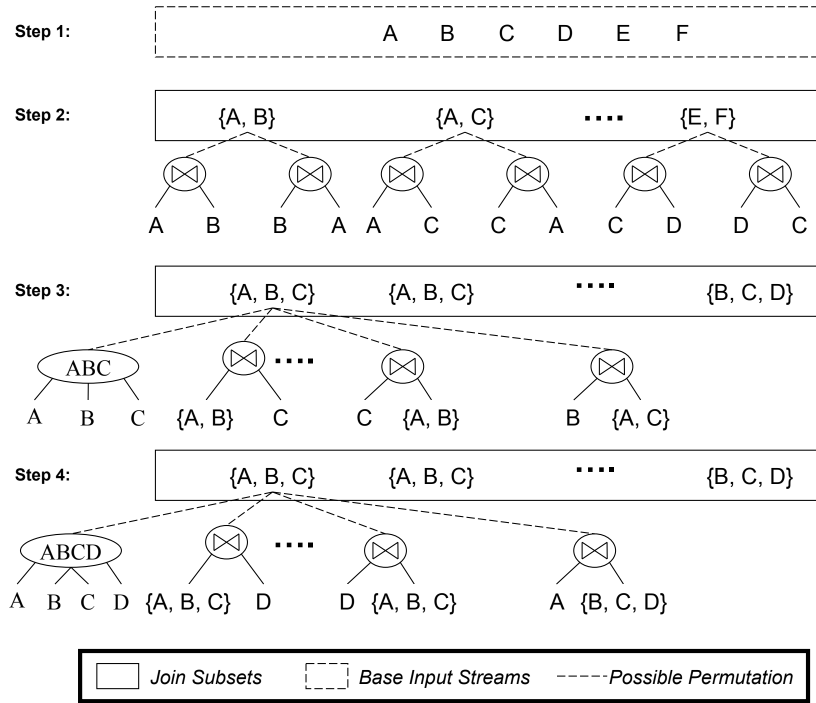


Figure 6: Steps In JTree-Finder For 6-way Join

Second, *JTree-Finder* makes use of the two-dimensional **cost model** described in Section 2 which calculates both CPU and memory resource utilizations. While traditional optimizers [24, 5, 2] instead only use the one-dimensional CPU or IO processing costs. The dual-constraints cost model enables us to prune sub-plans whose memory or CPU usage is greater than their respective system thresholds at an early stage.

Lastly, **termination condition**. *JTree-Finder* terminates either by 1) returning a positive result in the form of a set of qualified plans to choose from, or 2) returning a negative result when the available system resources are too limiting and thus no qualified plan exists in the solution space. When the available system resources are too restrictive, the algorithm can terminate early without having to explore the full search space. More precisely, the iterative process of the algorithm can terminate at some iteration k, where $k < n$, if no sub-plans of size k are found to be qualified.

Algorithm 1 JTree-Finder For Exploring JTree Solution Space

Input: Join Graph *JG* over $\mathcal{U}$ = $\{A, B, C\ldots\}$; Dual-Resource Constraints: CPU_{avail}, Mem_{avail}
Output: Set of Qualified Query Plans ($\mathcal{Q}$) or -1
1: *SubPlans*[ ]=$\emptyset$; //$SubPlans[k]$: set of plans of size k
2: Add all input streams in $\mathcal{U}$ to *SubPlans*[1]
3: **for** pSize = 2 to $|\mathcal{U}|$ **do**
4:   **for** all $V \subseteq \mathcal{U}$ and $|V|$ = pSize **do**
5:     $PT \leftarrow$ Find all distinct partitions of V
6:     **for** each partition $PT[i] \in PT$ **do**
7:       $P = Generate_Plans(PT[i], SubPlans[\,])$
8:       **for** each $\mathcal{P}_i \in P$ **do**
9:         **if** $(Memory(\mathcal{P}_i, JG) \leq Mem_{avail})$ AND $(CPU(\mathcal{P}_i, JG) \leq CPU_{avail})$ **then**
10:           **if** $pSize = |\mathcal{U}|$ **then**
11:             Add $\mathcal{P}_i$ to $\mathcal{Q}$
12:           **else**
13:             Add $\mathcal{P}_i$ to *SubPlans[pSize]*
14:       **if** $(SubPlans[pSize] = \phi)$ **then return** -1
15: **return** $\mathcal{Q}$ // Return Qualified Plans

Algorithm 1 presents the pseudo-code of JTree-Finder. Given a multi-join over a set of input streams, $\mathcal{U} = \{A, B, C, D, E, F\}$ and represented by a join graph JG. The available system resources are CPU_{avail} and Mem_{avail}. In the kth iteration of the algorithm, all combinations of input streams of size k over the set $\mathcal{U}$ are considered (Line: 4). V= $\{A, B, C, D\}$ is one such combination of size 4. We then partition V into pairwise disjoint subsets (of V) whose union is V. The total number of partitions for V is given by the *Bell number* (B_k), where $k = |V|$. For V = $\{A, B, C, D\}$, the various partitions (PT) are $\{(D), (A, B, C)\}$, $\{(A, B), (C, D)\}$, $\{(A), (B), (C), (D)\}$, etc. A partition $PT[i] \in PT$ is made up of subsets whose size is less than k. Partition $\{(D), (A, B, C)\}$ of V is made of two subsets (D) and (A, B, C). All possible implementations of subplans of size less than k have already been generated in the previous iterations. The algorithms then generates all possible query plans for the partition $\{(D), (A, B, C)\}$. We defined a function called $Generate_Plans$ (Line: 7) that takes as input arguments, a partition $PT[i]$ and all query plans generated in the previous iterations ($SubPlans[\,]$) and returns

a set of valid query plans that implement the partition $PT[i]$. Query plans of the partition $\{(D), (A, B, C)\}$ whose CPU or memory consumptions are within their respective available system resources are retained and added to $SubPlans[k]$ (Line: 13). While the plans whose CPU or memory resource consumptions are above their respective thresholds are pruned. If a qualified plan exists in the solution space, JTree-Finder is guaranteed to generate it in the nth iteration.

The algorithm returns a set of qualified plans found in the search (Line: 15). The choice of which qualified query plan to use is dependent on the end-applications. A few such decision policies are: 1) randomly choose one qualified query plan, 2) choose a query plan that is minimal in one of the resource usages (either CPU or memory), or 3) choose any one query plan that is not dominated with respect to both resource consumptions. Other choices among qualified plans may also be worth considering in special context, such as when handling multiple user queries, which we leave for future work (Section 8).

Time Complexity: Traditional optimizers [2] that use dynamic programming to generate optimal plans have an exponential time complexity [5]. Similarly, JTree-Finder has an exponential time-complexity and therefore is not practical for stream query processing

4. Evaluation of JTree-Finder

4.1. Experimental Methodology

Objectives. The goals of the first experiment are: 1) Verify the cost analysis used (Section 2) by comparing the performance of best mjoin and bjtree plans against the general jtree-based plans generated by *JTree-Finder* under different resource constraints, and 2) show that the inclusion of jtrees in the solution space increases the possibility of finding a qualified plan.

Parameter	Set 1	Set 2	Set 3	Set 4
M_a (MB)	300	300	30	30
λ_A (tuples/sec)	20	20	10	20
λ_B (tuples/sec)	20	20	10	20
λ_C (tuples/sec)	20	50	10	20
λ_D (tuples/sec)	—	—	10	20
σ_{AB}	0.05	0.02	0.1	0.02
σ_{BC}	0.5	0.5	0.15	0.2
σ_{CD}	—	—	0.1	0.05
W (ms)	5000	15000	15000	50000

Table 2: Parameter Configurations

Environment. The proposed plan generation framework is implemented in the continuous query processing system, [26]. Algorithms proposed in this work were implemented in Java. All experiments are conducted on an Intel 1.5 GHz machine with a 512 MB.

Organization: Our experiments are categorized into four sets by adjusting 1) availability of system resources CPU_{avail} and Mem_{avail} and 2) input stream arrival rates and selectivities. Our aim is to reproduce the four distinct scenarios presented in Section 3.1 (Figures 5.a-d), where a qualified plan exists in the solution space. The experimental sets are: 1) memory and CPU resources are sufficient for executing both bjtree and mjoin (Figure 5.a), 2) where CPU resources are sufficient for executing the bjtree but insufficient for mjoin (Figure 5.b), 3) memory is configured such that it is sufficient for mjoin but not for bjtree (Figure 5.c), 4) both CPU and memory resources are insufficient for either bjtree or mjoin, hence needing to include general *jtrees* (Figure 5.d) in the solution space.

Experimental Setup: A continuous query processing engine is stable when it can process the tuples accumulated in the each of its queues with none to little delay. By the *cost-per-unit-time CPU metric*, we mean the CPU time required by the query plan to process the tuples accumulated in all of its queues within one unit time. In this work, this is denoted as CPU_{reqd}. Therefore for a stable query processing the required CPU (CPU_{reqd}) must be ≤ 1 unit time (implying $CPU_{avail} = 1$). In our query engine, the time unit is $seconds$ and thus $CPU_{avail} = 1\ sec$. If $CPU_{reqd} > 1\ sec$ then the system will be saturated, with more and more tuples accumulating in the input queues making it infeasible to keep up with the new tuples coming from input streams.

Table 3: CPU Cost (ms) of Basic Tuple Operations

C_j	C_i	C_d
2.2×10^{-3}	2.0×10^{-4}	2.0×10^{-4}

We measured the per tuple run-time average costs of join (C_j), insert (C_i) and delete (C_d) using our query engine [26] running on a machine with 1.5MHz processor and 516MB memory. The results are displayed in Table 3. Given the per-tuple cost metrics, for an mjoin, bjtree or jtree plan we compute the CPU-utilization by our cost model. The effect of different CPU availabilities is achieved by increasing or decreasing parameters, like window sizes, stream arrival rates, and join selectivities. The available memory resource Mem_{avail} is controlled by setting the maximum Java heap size. Table 2 lists the parameter settings in the 4 sets of experiments.

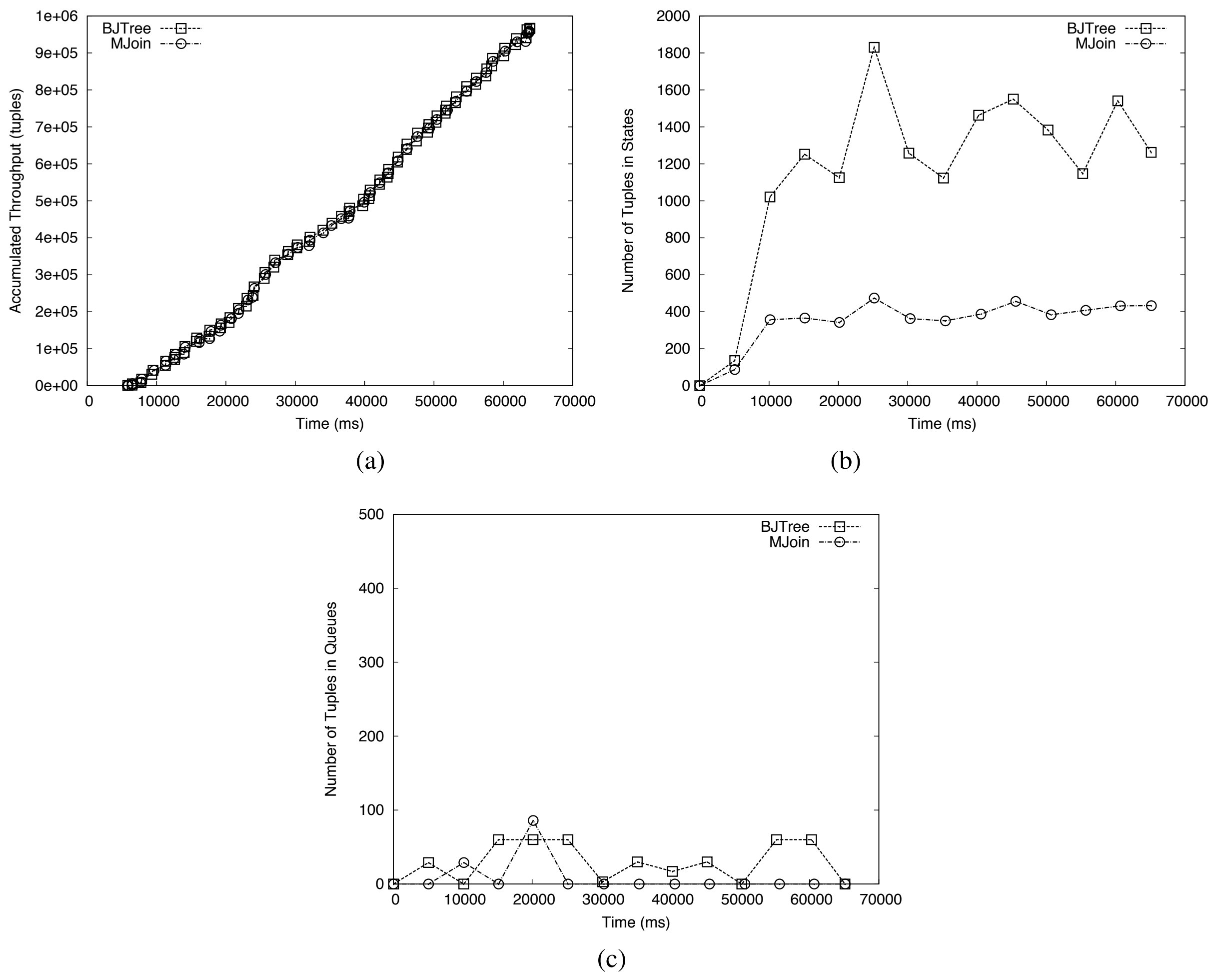


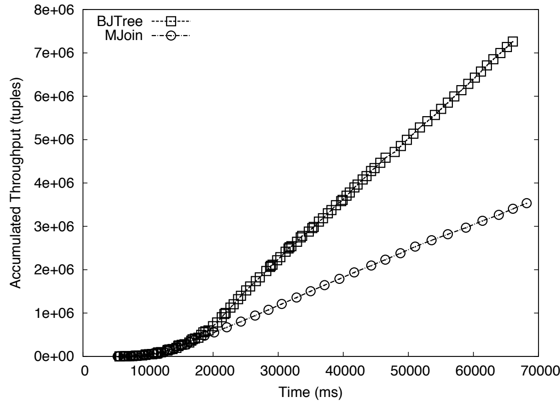
Figure 7: Experiment Set 1: (a) Accumulated Throughput (b) Tuples in States (c) Tuples in States Queues

4.2. Analysis of Different Plan Types

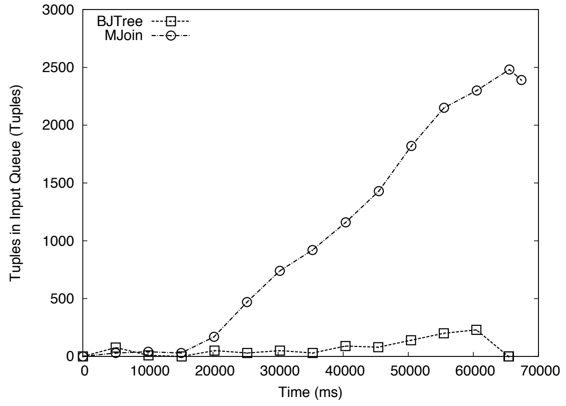
Experiment Set 1: When both mjoin and bjtree are qualified plans they have similar *accumulated throughput*[2] as shown in Figure 7.a. Sufficient CPU ensures that new tuples can be processed quickly without delay, which is true for both mjoin and bjtree, as in Figure 7.c. Figure 7.b clearly displays the much larger memory usage of bjtree for storing intermediate results in comparison to mjoin.

Experiment Set 2: In this scenario, the stream characteristics (input rate and selectivity) and window size make the best mjoin plan not qualified while the best bjtree plan be qualified. Figure 8.a shows that bjtree has a much higher throughput ($>$ 100% improvement)

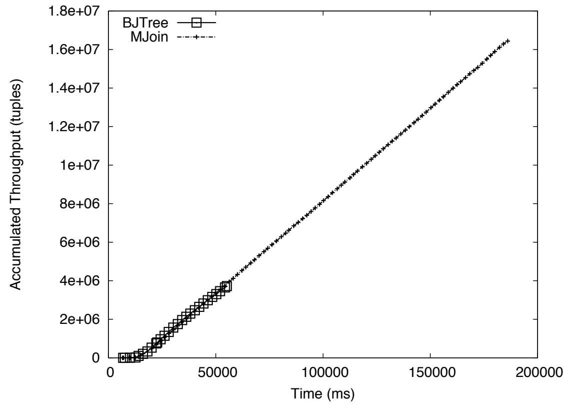
[2] *accumulated throughput*: total number of output tuples produced thus far.



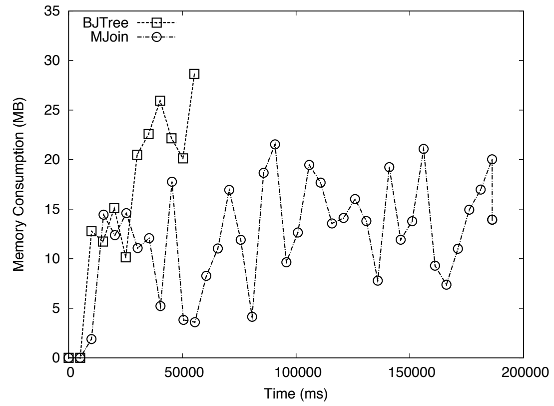
(a) Set 2: Accumulated Throughput



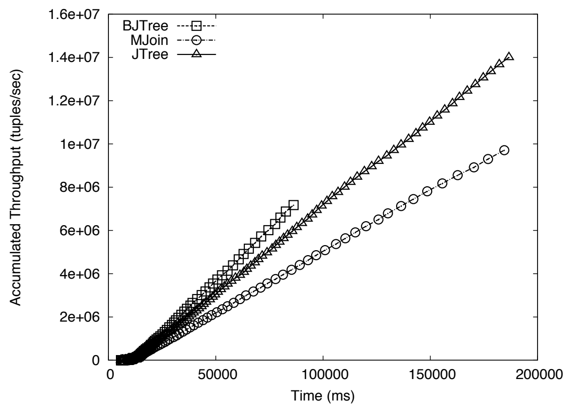
(b) Set 2: Tuples in Input Queues



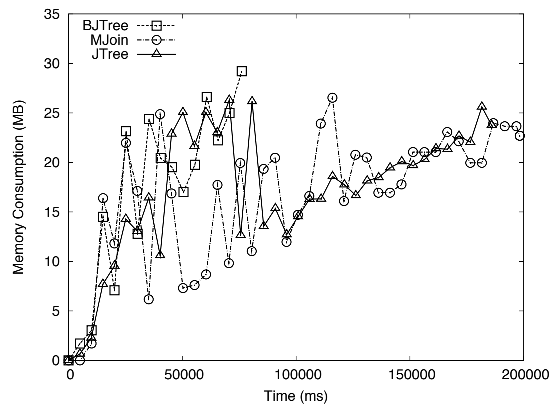
(c) Set 3: Accumulated Throughput



(d) Set 3: Memory Consumption



(e) Set 4: Accumulated Throughput



(f) Set 4: Memory Consumption

Figure 8: Experiment Set 2: (a and b), Experiment Set 3 : (c and d) and Experiment Set 4: (e and f)

than mjoin. Since mjoin is not qualified, new tuples cannot be processed right away and thus accumulate quickly in stream input queues (Figure 8.b), while bjtree processes new tuples right away keeping up with the input queues. Thus, the queue sizes are kept small on an average.

Experiment Set 3: When given enough CPU resources but limited memory, both mjoin and bjtree have similar throughput in the beginning (Figure 8.c). As the memory of bjtree accumulates (Figure 8.d) it reaches the threshold (M_a=30MB) at around 50,000ms. Once the memory threshold is reached the bjtree crashes due the lack of additional memory resources. In comparison, the memory consumed by mjoin is smaller and averages around 12MB after the start-up. Memory usage fluctuations in mjoin are due to memory temporarily used by the intermediate results.

Experiment Set 4: Neither mjoin nor bjtree are qualified plans, while a general jtree is found to be qualified. Figures 8.e and f compare the accumulated throughput and memory consumptions of the three plans. BJTree has the highest initial throughput. However, it quickly runs out of memory at around 80,000ms. Although jtree has a lower initial throughput than bjtree, it has a higher throughput than mjoin. Since jtree requires less memory than bjtree, it is able to continuously produce results.

In summary, the above results confirm our cost analysis, and demonstrate the need to extend the search space to also include general jtrees.

Comparative Study: In Section 6, we present an extensive comparative study of the proposed polynomial-time techniques by varying the number of streams **N in between 3 to 20 streams**.

5. Our Approach: Two-Layer Plan Generation

5.1. Overview

Qualified plan generation is an exponential time-complexity problem even with richer pruning techniques [25]. We therefore now put forth an efficient hill-climbing based **two-layer plan generation framework** that generates qualified query plans. Algorithm 2 presents the pseudo-code of our framework.

The **first-layer** utilizes the positive correlation between memory and CPU by transforming the two-dimensional problem into a one-dimensional minimization problem, this way focusing on minimizing one of the two cost metrics. This offers three benefits. First, it allows us to employ state-of-the-art algorithms to optimize mjoin [15, 9, 19] (Line: 2) and bjtrees [5, 1] (Line :7). Second, the generated mjoin is guaranteed to utilize the minimum memory resources of all possible plans, while the generated bjtree has the least amount of CPU usage as no re-computation is required. By piggybacking on the mjoin and bjtree plans generated by the first-layer we identify starting points for our hill-climbing algorithm in our second-layer of optimization. Third, early termination can often be achieved,

namely when either a qualified plan is found (Line: 5, 10), or when the minimal bounds on the required system resources are larger than their respective system thresholds (Line: 6, 11). For example, consider the scenario when the memory resources needed by the mjoin, Memory($\mathcal{P}_{mjoin}, JG$) are larger than the available memory, Mem_{avail}. In such cases no plan exists in the search space with a lesser memory utilization than $\mathcal{P}_{mjoin}$. If the first-layer is unable to return a qualified plan yet does not determine the in-feasibility of the problem, the second-layer is triggered to explore the jtree search space.

Algorithm 2 Two-Layer Plan Generation

Input: Join Graph *JG*, $S = \{A, B, C \ldots\}$ Resource Constraints: CPU_{avail}, Mem_{avail}
Output: Qualified Query Plan $\mathcal{P}$ or -1
1: // **Layer-One**
2: $\mathcal{P}_{mjoin}$ = Find_MJoin_Ordering(JG)
3: **if** Memory($\mathcal{P}_{mjoin}, JG) \leq Mem_{avail}$ **then**
4:     **if** CPU($\mathcal{P}_{mjoin}, JG) \leq CPU_{avail}$ **return** $\mathcal{P}_{mjoin}$
5: **else return** -1 // Mem_{avail} Bound Restrictive
6: $\mathcal{P}_{bjtree}$ = Find_BJTree_Ordering(JG)
7: **if** CPU($\mathcal{P}_{bjtree}, JG) \leq CPU_{avail}$ **then**
8:     **if** Memory($\mathcal{P}_{bjtree}, JG) \leq Mem_{avail}$ **return** $\mathcal{P}_{bjtree}$
9: **else return** -1 // CPU_{avail} Bound Restrictive
10: // **Layer-Two**
11: $\mathcal{P} = State_Selection(JG, CPU_{avail}, Mem_{avail}, \mathcal{P}_{mjoin})$
12: $Mem_r = Memory(\mathcal{P}, JG); CPU_r = CPU(\mathcal{P}, JG)$
13: **if** ($Mem_r \leq Mem_{avail}$) AND ($CPU_r \leq CPU_{avail}$) **return** $\mathcal{P}$
14: $\mathcal{P} = State_Removal(JG, CPU_{avail}, Mem_{avail}, \mathcal{P}_{bjtree})$
15: $Mem_r = Memory(\mathcal{P}, JG); CPU_r = CPU(\mathcal{P}, JG)$
16: **if** ($Mem_r \leq Mem_{avail}$) AND ($CPU_r \leq CPU_{avail}$) **return** $\mathcal{P}$
17: **return** -1 // No Qualified Plan Found

Intuitively, the aim of the second layer is to decrease one cost factor while increasing the other, until both are under the system thresholds. In this regard, we present two polynomial-time algorithms that start from either $\mathcal{P}_{mjoin}$ or $\mathcal{P}_{bjtree}$ plan generated in the previous layer. Our *state-selection* algorithm (Section 5.2) starts with an mjoin plan $\mathcal{P}_{mjoin}$ and selects intermediate states to be materialized. In contrast, our *state-removal* algorithm (Section 5.3) starts with a bjtree $\mathcal{P}_{bjtree}$ and selects smaller binary joins to be merged into a single mjoin. In either case, the result is a **jtree**. The choice of which algorithm is dependent on several factors such as where $\mathcal{P}_{mjoin}$ and $\mathcal{P}_{bjtree}$ lie in the solution space (Figure 5.d). If no qualified plan can be generated by the second-layer of plan generation, the only alternative is to invoke more radical adaptation techniques, such as memory-spilling [11, 12, 9], load shedding [8, 10] or query distribution [27, 28].

In summary, the plan generation framework will have one of four possible results: (1) a qualified mjoin, (2) a qualified bjtree, (3) a qualified jtree, or (4) a negative result

indicating that the system resources are not sufficient. The first-layer either outputs (1), (2) or (4), or triggers the second-layer. The second-layer either identifies a qualified jtree (3), or terminates without a result (4).

5.2. The State-Selection Algorithm

We now describe our *state-selection algorithm* that iteratively selects intermediate results to be stored thereby saving valuable CPU resources otherwise spent on re-computation. The pseudo-code for the algorithm is presented in Algorithm 3.

The algorithm takes as input arguments a user query represented by its join graph JG, system resource constraints CPU_{avail} and Mem_{avail} and the mjoin plan $\mathcal{P}_{mjoin}$ generated by the first layer of plan generation. Mjoin plan only stores the tuples from each of the input streams in states, thus $\mathcal{P}_{mjoin}$ has the least memory utilization. However, since the mjoin plan has to recompute all intermediate results, the CPU resource usage needed to recompute the intermediate tuples is high. In this technique, we apply the principle of **negative correlation** of sacrificing memory resources by storing some intermediate results, thereby saving on the corresponding re-computation CPU costs.

Selecting which intermediate state(s) to be stored can be viewed as selecting edges in a join graph. The choice of which intermediate states to store is determined by two factors: *edge frequency* and *edge weight*. The *edge frequency* is the number of times an edge appears in the join sequences. In Figure 9.b, for input stream A the first layer of optimization generates the best possible join order (in polynomial time) to process the input tuples from A, similarly we have a join ordering for the input stream B, and so on. In Figure 9.b, the edge between input stream A and C appears 6 times in the set of join orders, while the edge between D and C appears twice. Therefore, storing the intermediate results of $A \bowtie C$ would be of more benefit than storing results of $D \bowtie E$. Thus, heuristically, the higher the edge frequency, the more likely it is that storing the corresponding intermediate results can save CPU costs.

The *edge weight* connecting vertices V_x and V_y is the estimated intermediate state size, $\lambda_{V_x}\lambda_{V_y}\sigma_{V_xV_y}$. The aim of the algorithm is to reduce the CPU resource consumption by using the least amount of memory resources. Therefore, the *state selection* algorithm chooses the edge with the largest (frequency/weight) ratio.

To illustrate, for the join graph in Figure 9.a and the join ordering in Figure 9.b we can derive the edge frequency as described in Figure 9.c. In each iteration the algorithm selects an edge with the $max(\frac{frequency}{weight})$ ratio as the next state to be stored.

For each edge $\overline{V_xV_y}$, the algorithm accesses the benefits of storing its intermediate states. Vertices V_x and V_y are merged into one vertex with input rate of $\lambda_{V_x}\lambda_{V_y}\sigma_{V_xV_y}$, i.e., we create a binary join $V_x \bowtie V_y$ whose output is now fed to the mjoin operator. Now, for the mjoin with one less input stream, we employ the popular polynomial-time mjoin

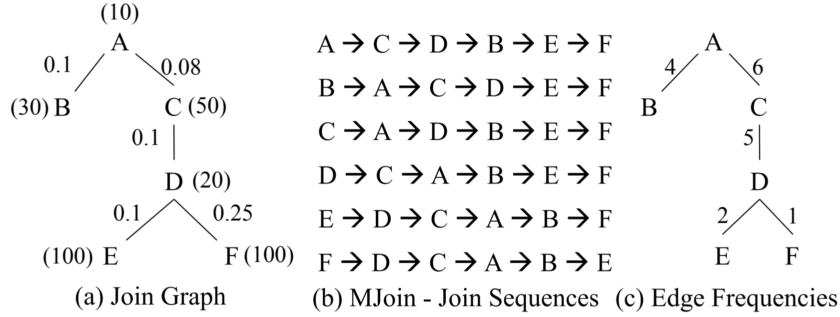


Figure 9: Counting Edge Frequencies

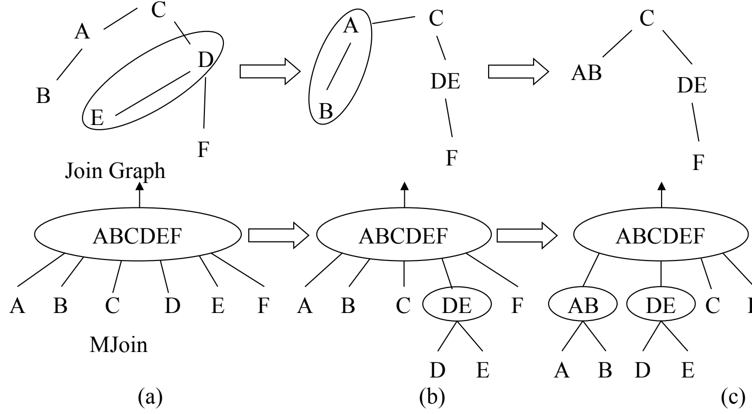


Figure 10: Generating JTree By State Selection

ordering algorithm [15, 19, 9] to generate the best possible join ordering for each of its input streams. If CPU costs of the new mjoin are smaller than the current plan, the state $S_{V_xV_y}$ is stored. The algorithm proceeds iteratively until: (1) no intermediate states can further be stored to reduce the CPU usage, or (2) memory usage of jtree exceeds Mem_{avai}.

To elaborate, let the edge $\overline{DE}$ in Figure 10.a be the candidate edge to be merged. The mjoin with 6 input streams is now transformed into an mjoin with 5 inputs where one of its inputs is the results from the binary join $D \bowtie E$ (Figure 10.b). The output rate of the merged vertices DE is $\lambda_D \lambda_E \sigma_{DE}$.

The process described so far ends up generating binary join nodes for the merged vertices. We now add further steps to merge several binary joins into an mjoin. This is done only when both memory and CPU costs can be saved. Let the edge between vertex C and DE in Figure 11.a be chosen as the next edge to be merged. The new merged vertex can either be implemented as a bjtree or a single mjoin as shown in Figures 11.b and c respectively. In the state-selection algorithm, we define a function called $Optimize$ (Line: 9) that takes as inputs the merged vertex, the modified join graph and the system resource

Algorithm 3 State Selection Algorithm

Input: Dual-Resource Constraints: CPU_{avail}, Mem_{avail}; Join Graph JG; $\mathcal{P}_{mjoin}$
Output: Qualified JTree Plan $\mathcal{P}$ or -1

1: $\mathcal{P} = \mathcal{P}_{mjoin}$
2: $CPU_r = CPU(\mathcal{P}, JG)$; $Mem_r = Memory(\mathcal{P}, JG)$
3: **while** $(CPU_r > CPU_{avail}) AND (Mem_r < Mem_{avail})$ **do**
4: E = Set of candidate edges in JG
5: **while** (E != null) **do**
6: Choose $E_i \in E$ with max($\frac{frequency}{weight}$)
7: Let E_i be an edge that connect vertices V_x and V_y
8: JG_{new} = Merge vertices V_x and V_y to V_{xy} in JG
9: $Optimize(V_{xy}, JG_{new}, CPU_{avail}, Mem_{avail})$
10: $\mathcal{P}_{new} \leftarrow$ MJoin_Ordering(JG_{new})
11: CPU_{new} = CPU($\mathcal{P}_{new}, JG_{new}$)
12: **if** $(CPU_{new} < CPU_r)$ **then**
13: $\mathcal{P} = \mathcal{P}_{new}$; CPU_r = CPU_{new}; $JG = JG_{new}$
14: **if** $Memory(\mathcal{P}, JG_{new}) > Mem_{avail}$ **then**
15: **return** -1; // No qualified plan found
16: Remove E_i from E
17: **if** ($|E|$ == 0) **then return** -1
18: **return** $\mathcal{P}$ // Exist loop when qualified plan found

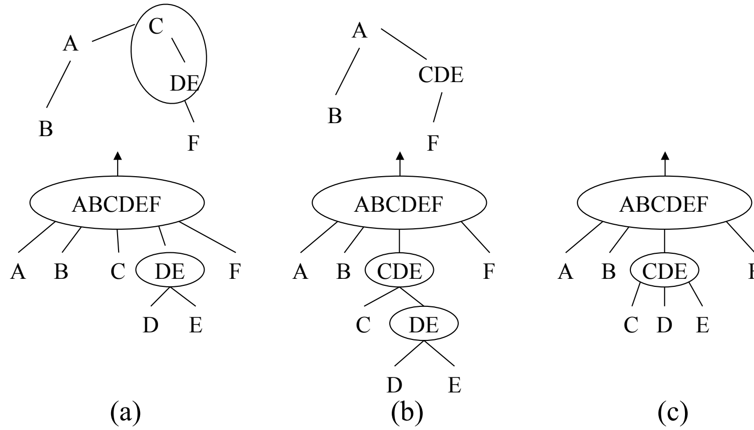


Figure 11: Operator Breaking and Merging

constraints to return a qualified implementation of the merged vertex.

Time Complexity: A join graph with n streams has at most $n(n-1)/2$ edges. After merging an edge, we select a state-of-the-art ordering algorithm [9, 15] having a time-complexity of $\mathcal{O}(n^2 log(n))$ to recompute the join sequences. Thus *state-selection* has a worst case time-complexity of $\mathcal{O}(n^3 log(n))$.

5.3. The State-Removal Algorithm

We now present the *state-removal* algorithm that aims to save precious memory resources by removing intermediate states and merging the join operators into an mjoin operator that does not store intermediate states. The pseudo-code for the algorithm is presented in Algorithm 4. The algorithm takes as input arguments a user query represented by its join graph JG, system resource constraints CPU_{avail} and Mem_{avail} and the bjtree plan $\mathcal{P}_{bjtree}$ generated by the first layer of plan generation. This bjtree plan has the least amount of CPU resources usage as it does not perform any re-computation of intermediate results as in an mjoin plan. However, the memory resources needed to store these intermediate results is very high. Therefore, this technique of *state removal* aims to use the principle of **negative correlation** and removes some intermediate results thereby saving on the memory resources utilized but in turn increasing the total CPU consumption.

Figure 12 illustrates the application of the state-removal techniques for a bjtree with four intermediate states (represented by a rectangle in Figure 12.a). To remove the intermediate state at join operator BC the algorithm merges operators AB and BC into a single mjoin operator ABC (as in Figure 12.b). A state-of-the-art mjoin ordering algorithm [15, 19, 9] is then used to find the join orderings for the new mjoin operator ABC. This process is repeated until a qualified jtree is found, or all states have been explored.

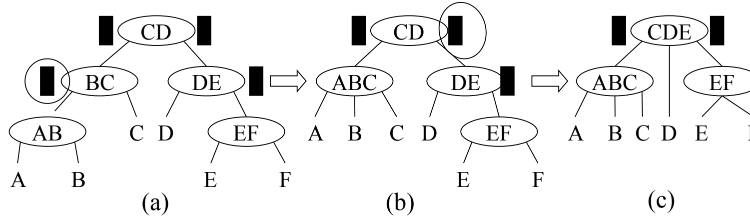


Figure 12: Removing State by Merging Joins.

For each candidate intermediate state to be removed, two factors must be considered, namely, the memory saved by removing the state and the additional CPU resources needed to recompute the intermediate results. If $CPU_{current}$ and CPU_{new} represent the CPU resource utilization of the current plan and the new candidate plan after operator merging respectively, the increase in CPU costs can be computed as $\Delta_{CPU} = CPU_{new} - CPU_{current}$. The memory saved is in fact the size of the removed intermediate state. *State quality ratio* is defined as $\Delta_{CPU}/|S[i]|$. Intuitively, in each iteration of the algorithm the intermediate state with the smallest state quality ratio is removed. The intermediate states are removed by merging two corresponding join operators into a larger mjoin operator. In scenarios when the CPU costs of maintaining intermediate states surpasses the cost of recomputing the state, the Δ_{CPU} factor is negative.

Algorithm 4 State Removal Algorithm

Input: Dual-Resource Constraints: CPU_{avail}, Mem_{avail}; Join Graph JG; $\mathcal{P}_{bjtree}$
Output: Qualified JTree Plan $\mathcal{P}$ or -1

1: $\mathcal{P} = \mathcal{P}_{bjtree}$
2: S = set of all intermediate states in $\mathcal{P}_{current}$
3: **while** $S \neq \phi$ **do**
4: CPU_r = CPU($\mathcal{P}, JG$); min_ratio = ∞
5: **for** (each $S_i \in S$) **do**
6: $op1$ = join operator contains S_i
7: $op2$ = join operator with feeds tuples to S_i
8: $\mathcal{P}_{new} = Merge(\mathcal{P}, op1, op2)$
9: // 1. Merges $op1$ and $op2$ into $newOp$;
10: // 2. Generates join ordering for $newOp$
11: CPU_{new} = CPU($\mathcal{P}_{new}, JG$)
12: $state - ratio = (CPU_{new} - CPU_r)/(|S_i|)$
13: **if** ($state - ratio < min_ratio$) **then**
14: $min_ratio = state - ratio$; $state$ = S_i; $\mathcal{P}_s$ = $\mathcal{P}_{new}$
15: Remove $state$ from S; $\mathcal{P} = \mathcal{P}_s$
16: **if** (CPU($\mathcal{P}, JG$) $\leq CPU_{avail}$) AND (Memory($\mathcal{P}, JG$) $\leq Mem_{avail}$) **then**
17: **return** $\mathcal{P}$ // A Qualified Plan
18: **return** -1 // No Qualified Plan Found

Time Complexity: A join graph with n vertices has at most $n - 1$ intermediate states. Therefore, state selection and operator merging process may be repeated at most $n - 1$ times. If the chosen state-of-the-art mjoin ordering algorithm [15, 9, 19] has the time-complexity of $\mathcal{O}(n^2 log(n))$, the total running time is bounded by $\mathcal{O}(n^3 log(n))$.

6. Comparative Study

6.1. Experimental Methodology

Objectives. The goals of this second experimental evaluation include the comparison of: 1) the average time required by the proposed algorithms to generate a qualified plan, and 2) the *effectiveness* of the proposed algorithms to find a qualified query plan if one such plan exists in the solution space. 3) the memory and CPU resource utilization of the qualified plans generated by the *state-selection* versus by the *state-removal* algorithms.

Environment. We use the same testing environment that was presented in Section 4.

Organization. To determine the effectiveness of the proposed algorithms and observe trends we varied the number of input streams N from 3 to 20. For each N, the setup process involves: 1) randomly generate an input rate for each stream within the range of [1, 100] tuples/second, 2) joins among input streams are randomly selected, and 3)

the corresponding join selectivities are randomly generated within the range of (0,1)[3]. This setup process is repeated 100 times for each N. Therefore, a total of (20-3+1)*100 = 1800 different parameter settings, which is a sufficiently large sample set to illustrate performance trends studied. The CPU threshold defined as the maximum amount of time needed to process the tuples arriving within the time unit of 1 second, thus CPU_{avail} = 1 second.

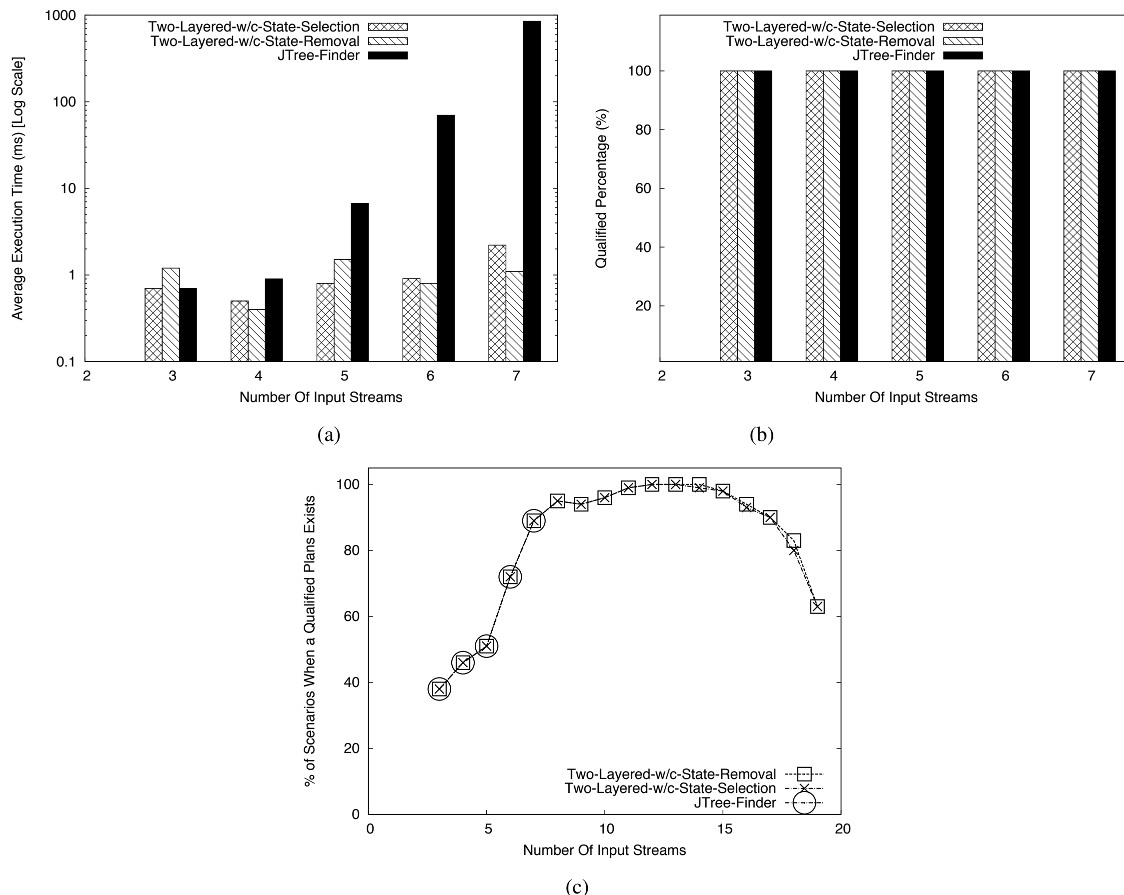


Figure 13: a) Avg. Execution Time b) Qualified Percentage c) % Of Scenarios when a Qualified Plan Exists

6.2. Effectiveness of Proposed Algorithms

Figure 13 compares the average execution times needed for our proposed algorithms to generate a qualified plan. *JTree-Finder*, due to its exponential time complexity is much slower in comparison to our polynomial time strategies, and runs out of resources for a

[3] $Join\ selectivity = (num\ of\ outputs)/(num\ of\ possible\ outputs)$

smaller N. Since the *JTree-Finder* algorithm is searching the much larger qualified plan solution space, it can only support a smaller N that other state-of-the-art dynamic programming techniques [5, 24]. Also, all algorithms are implemented in Java.

Definition 4. *Given X different experimental settings where a qualified plan exists in the solution space, if an algorithm finds a qualified query plan in Y ($Y \leq X$) such settings, the qualified percentage of the algorithm is* $(Y/X * 100)$.

The effectiveness of an algorithm is quantified by its *qualified percentage*. Figure 13.b illustrates the effectiveness of the proposed algorithms. JTree-Finder is guaranteed to find a qualified plan by searching the entire general jtree solution space, if one such plan exists. We observe that our heuristic-based *state-selection* and *state-removal* algorithms have identical qualified percentage (=100%) to JTree-Finder for various N. Comparison of our heuristic-based algorithms cannot be done for larger N as *JTree-Finder* fails due to its exponential time-complexity.

We note that for some stream characteristics and system resource threshold settings a qualified plan may not exist in the solution space. This is reflected in Figure 13.c where JTree-Finder is unable to find a qualified plan– given that JTreeFinder explores the full search space, this implies that no solution exists. No trends can be drawn from Figure 13.c as it merely reflects the state of the particular solution space we studied.

6.3. Resource Utilization of Generated Plans

Lastly, we compare the memory and CPU usages of the qualified plans generated by *state-selection* and *state-removal*. Figures 14 a, b and c depict the distributions of memory and CPU costs for all qualified query plans found for N equal to 3, 5 and 10, respectively. Qualified plans generated by state-selection generally tend to have smaller memory costs but larger CPU costs as compared to the qualified plans generated by state-removal. This trend becomes apparent as N increases. In Figure 14.a when N = 3, the qualified plans from both algorithms are mixed on the plot. As N increases, the qualified plans generated by state-selection tend to be located more at the upper-left area, while the qualified plans generated by state-removal tend to be located more at the lower-right area. The two sets of qualified plans are clearly separated from each other. This is due to in part their differences in the starting points of the two strategies. An mjoin usually has smaller memory costs but larger CPU costs than a bjtree for the same query. Optimizing from each starting point has the tendency to reach a qualified plan that is closer to that starting point.

6.4. Discussion: Statistic Gathering and Plan Migration

The focus of this work is to design efficient algorithms for finding qualified continuous query plans at runtime. We employ existing methods [29, 15] to collect statistics at

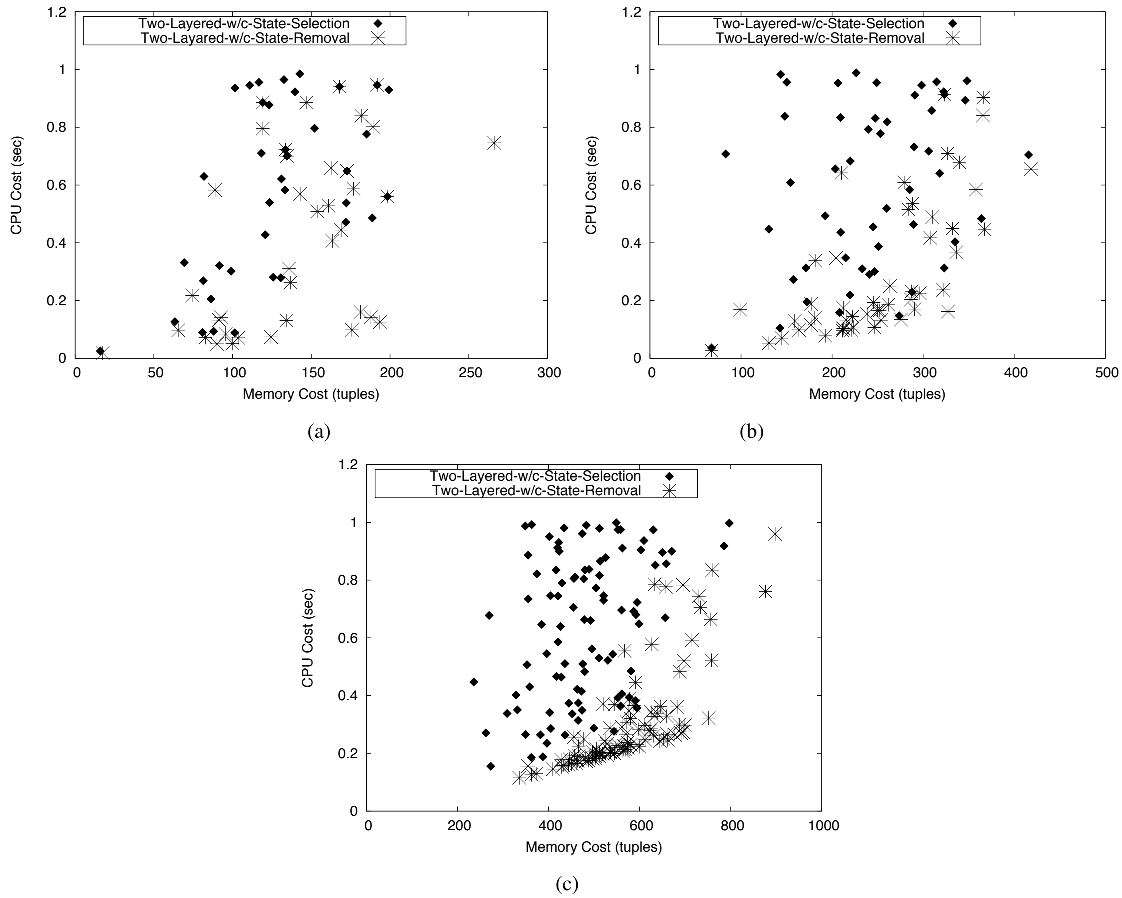


Figure 14: Distribution of Qualified Plans: a) N = 3 b) N = 5 and c) N = 10

runtime. When a better plan is found by any one of our proposed algorithms, we apply migration strategies [30, 31] to safely transfer the current query plan to the new plan. These migration strategies which focus on binary join plans can be easily extended to handle the movement of join states across our general jtree plans.

7. Related Work

The growing number of on-line users as well as the increased usage of sensors and RFID networks in various real-world scenarios has increased the availability of data streams on the web [32]. This phenomenon has resulted in the development of alternative streaming database systems [33, 34, 35, 36, 37]. The CQL continuous query language by A. Arasu et al [38] is a general SQL-based language proposed to express continuous queries against streams and updatable relations.

State-of-the-art techniques [1, 2, 9, 15, 19] have focused on the optimization of a single cost function by exploiting the positive correlation between memory and CPU usages. However, qualified plan generation has largely been ignored. Multi-join query optimization in static databases has focused on bjtrees [1, 2, 3, 5]. In contrast, the mjoin operator [9, 19] is popular in streaming databases, supported by several heuristic-based ordering algorithms [15, 19]. After inventing the mjoin for stream processing, [9] acknowledges the problem of excessive re-computation costs of intermediate states. While [9] suggests that a large mjoin with ≥ 5 streams may need to be split into two smaller mjoins, they do not concern themselves with memory constraints. Their experiments all deal with extremely small join selectivities ($\sigma = 0.0005\%$) and therefore the total number of intermediate results are relatively small in their experiments – thus memory overflow is not dealt dealt with in their work. Therefore, [9] does not tackle the open problem of qualified plan generation - the focus of our work. The Eddy approach [7] simulates an mjoin operator by introducing the STEM structure to enable dynamic join ordering at the tuple granularity. However, like mjoin, it suffers from the additional re-computation CPU costs.

The *A-Caching* algorithm [29] optimizes a single mjoin operator by adding/removing temporary caches for certain intermediate results. Our proposed algorithms differ significantly from A-Caching in several aspects. First, A-Caching only deals with equi-join predicates, relying on value-based hashing to detect a cache hit/miss. Our two-layer plan generation framework is general and can work with any physical implementation of multi-join queries. Second, A-Caching restricts the problem space by only considering solutions with a set of *non-overlapping* caches. In other words, two caches cannot have common joins. Our proposed algorithms do not have this restriction, thus broadening the range of intermediate results that can be simultaneously stored. Lastly, A-Caching is a novel operator-level join implementation strategy of a single mjoin operator. Our solution instead looks at the query plan-level, exploring the entire jtree solution space.

Handling multiple resources has been studied in distributed systems for parallel query optimization and task scheduling [39, 17, 16, 40, 14]. Parallel optimization proposed in [39] minimizes CPU costs at compile time, and delays the decision on buffer size to run-time. At run-time, based on observed cardinalities, one of the sub-plans is chosen. In contrast, our algorithms considers both CPU and memory costs factors when generating qualified query plans. Mariposa [14] optimizes a parallel query plan based on a concrete user-defined cost-delay curve. In our work, we do not assume that the relationship between CPU and memory is given *a priori*. In fact, in practice this is hard to capture, as elaborated in Section 1. [40] distributes operators of a plan to several processors while exploring the trade-offs among multiple system resources. Our work instead finds a plan that satisfies multiple resource constraints within a central system.

8. Conclusions and Future Work

In this work, we recast that continuous query plan generation, no longer a minimization problem as typically approached by state-of-the-art techniques, but rather as a constraint satisfaction problem. For this, we expand the traditional search space of pure mjoin- or pure bjtree-based query plans to include general jtrees. We explore the trade-offs and correlations between CPU and memory usage functions, both positive and negative. In this effort, we first present our *JTree-Finder* algorithm that is guaranteed to generate a qualified plan. Second, to provide an efficient run-time solution we present a two-layered plan generation framework. In our framework, we propose two polynomial-time algorithms *state-selection* and *state-removal* exploit the correlations to generate a qualified plan. Our experimental evaluation using an existing continuous query engine verifies our cost model and measures the CPU and memory resource usages of the generated plans. The experimental results clearly demonstrate: 1) the need to search the entire solution space including general jtrees so not to miss potentially qualified plans and 2) the effectiveness of our algorithms over the state-of-the-art techniques.

Many directions for future work are possible. We could consider the constraint problem for handling multiple queries. Two alternatives for addressing this problem may be to: 1) first allocate a fixed amount of the CPU and memory resources to each query, and then to directly apply the strategies presented in this work to each of the queries, or 2) to simultaneously build the plans for the different queries such that their collective resource consumptions are within bounds. While the former alternative can be easily achieved, the latter may be more practical as the criteria for pre-allocation of resources is a challenging problem in itself. Another avenue for future work is to add additional constraints such as latency and then to identify their correlations to CPU and memory resource usages.

References

[1] T. Ibaraki, T. Kameda, On the optimal nesting order for computing n-relational joins, TODS 9:3 (1984) 482–502.

[2] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, in: SIGMOD, 1979, pp. 23–34.

[3] A. N. Swami, B. R. Iyer, A polynomial time algorithm for optimizing join queries, in: ICDE, 1993, pp. 345–354.

[4] Y. E. Ioannidis, Y. C. Kang, Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization, in: SIGMOD, 1991, pp. 168–177.

[5] D. Kossmann, K. Stocker, Iterative dynamic programming: a new class of query optimization algorithms., TODS 25:1 (2000) 43–82.

[6] M. Steinbrunn, G. Moerkotte, A. Kemper, Heuristic and randomized optimization for the join ordering problem, VLDB J. 6:3 (1997) 191–208.

[7] S. Madden, M. A. Shah, J. M. Hellerstein, V. Raman, Continuously adaptive continuous queries over streams, in: SIGMOD, 2002, pp. 49–60.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. B. Zdonik, Monitoring streams - a new class of data management applications, in: VLDB, 2002, pp. 215–226.

[9] S. Viglas, J. F. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in: VLDB, 2003, pp. 285–296.

[10] A. Ayad, J. F. Naughton, Static optimization of conjunctive queries with sliding windows over infinite streams, in: SIGMOD, 2004, pp. 419–430.

[11] B. Liu, Y. Zhu, E. A. Rundensteiner, Run-time operator state spilling for memory intensive long-running queries, in: SIGMOD, 2006, pp. 347–358.

[12] T. Urhan, M. J. Franklin, Xjoin: A reactively-scheduled pipelined join operator, IEEE Data Eng. Bull. 23 (2) (2000) 27–33.

[13] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in: VLDB, 2003, pp. 309–320.

[14] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu, Mariposa: A wide-area distributed database system, VLDB J. 5 (1) (1996) 48–63.

[15] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Adaptive ordering of pipelined stream filters., in: SIGMOD, 2004, pp. 407–418.

[16] C. H. Papadimitriou, M. Yannakakis, Multiobjective query optimization, in: PODS, 2001, pp. 52–59.

[17] W. Hasan, R. Motwani, Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism, in: VLDB, 1994, pp. 36–47.

[18] C. Monma, J. Sidney, Sequencing with series-parallel precedence constraints, in: Maths of Operations Research 4, 1979, pp. 215–224.

[19] L. Golab, M. T. Özsu, Processing sliding window multi-joins in continuous queries over data streams, in: VLDB, 2003, pp. 500–511.

[20] M. A. Hammad, M. J. Franklin, W. G. Aref, A. K. Elmagarmid, Scheduling for shared window joins over data streams., in: VLDB, 2003, pp. 297–308.

[21] L. Getoor, B. Taskar, D. Koller, Selectivity estimation using probabilistic models, in: SIGMOD, 2001, pp. 461–472.

[22] A. Deshpande, M. N. Garofalakis, R. Rastogi, Independence is good: Dependency-based histogram synopses for high-dimensional data, in: SIGMOD, 2001, pp. 199–210.

[23] J. Kang, J. F. Naughton, S. Viglas, Evaluating window joins over unbounded streams., in: ICDE, 2003, pp. 341–352.

[24] G. Moerkotte, T. Neumann, Dynamic programming strikes back, in: SIGMOD, 2008, pp. 539–552.

[25] S. Ganguly, W. Hasan, R. Krishnamurthy, Query optimization for parallel execution, in: SIGMOD, 1992, pp. 9–18.

[26] Reference omitted due to double blind reviewing.

[27] Y. Xing, S. B. Zdonik, J.-H. Hwang, Dynamic load distribution in the borealis stream processor, in: ICDE, 2005, pp. 791–802.

[28] F. Tian, D. J. DeWitt, Tuple routing strategies for distributed eddies, in: VLDB, 2003, pp. 333–344.

[29] S. Babu, K. Munagala, J. Widom, R. Motwani, Adaptive caching for continuous queries, in: ICDE, 2005, pp. 118–129.

[30] Y. Yang, J. Krämer, D. Papadias, B. Seeger, Hybmig: A hybrid approach to dynamic plan migration for continuous queries, TKDE. 19 (3) (2007) 398–411.

[31] Y. Zhu, E. A. Rundensteiner, G. T. Heineman, Dynamic plan migration for continuous queries over data streams, in: SIGMOD, 2004, pp. 431–442.

[32] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. A. Shah, Telegraphcq: Continuous dataflow processing for an uncertain world., in: CIDR, 2003.

[33] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. B. Zdonik, The design of the borealis stream processing engine, in: CIDR, 2005, pp. 277–289.

[34] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, M. F. Mokbel, Nile-pdt: A phenomenon detection and tracking framework for data stream management systems., in: VLDB, 2005, pp. 1295–1298.

[35] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: PODS, 2002, pp. 1–16.

[36] R. Avnur, J. M. Hellerstein, Eddies: Continuously adaptive query processing, in: W. Chen, J. F. Naughton, P. A. Bernstein (Eds.), SIGMOD, 2000, pp. 261–272.

[37] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, Niagaracq: A scalable continuous query system for internet databases, in: SIGMOD, 2000, pp. 379–390.

[38] A. Arasu, S. Babu, J. Widom, The cql continuous query language: semantic foundations and query execution, VLDB J. 15 (2) (2006) 121–142.

[39] W. Hong, M. Stonebraker, Optimization of parallel query execution plans in xprs, in: PDIS, 1991, pp. 218–225.

[40] M. N. Garofalakis, Y. E. Ioannidis, Multi-dimensional resource scheduling for parallel queries, in: SIGMOD, 1996, pp. 365–376.