GENERATING QUALIFIED PLANS FOR MULTIPLE QUERIES IN DATA STREAM SYSTEMS

by

Zain Ahmed Alrahmani

Submitted in partial fulfilment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Because stream processing applications operate under strict time constraints, most previous research focuses on enhancing real-time response by decreasing a single cost function. Although such research generates an optimal or near-optimal query execution plan under the selected cost function, either in terms of memory or CPU resources, the generated plan may not actually qualify for real execution; that is, although it is optimal in minimizing the chosen memory (CPU) cost function, it may exceed the available capacity of the CPU (memory) resource. A plan is not qualified if it is optimal or near-optimal in one resource usage, whereas it is out of bound in the other. These kinds of plans are not viable in stream processing applications because one of the resource usages, either CPU or memory, will hinder the system in processing queries. This thesis proposes a technique that generates qualified global plans for multiple queries under constraints for both CPU and memory resources by scheduling MJoin and BJtree operators while sharing common operations and their results among queries.

LIST OF ABBREVIATIONS USED

BJtree	Binary Join Tree
CPU	Central Processing Unit
DSMS	Data Stream Management System
JTree	Join Tree of MJoins and Binary Joins
LWO	Largest Window Only
MJoin	Multi-Join
SWF	Smallest Window First

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Peter Bodorik, for his continuous support of my research, patience, motivation, enthusiasm, and immense knowledge. His guidance helped me throughout the research and writing of this thesis.

In addition, I would like to extend my sincerest thanks and appreciation to Dr. Vlado Keselj and Dr. Michael McAllister for serving on my guiding committee and providing valuable suggestions to improve the quality of my work.

CHAPTER 1 : INTRODUCTION

The past several years have seen the emergence of applications that deal with data streams, which are referred to as data stream applications. In these applications, data are unbounded streams of values and are generated continually at very high rates. In other words, the stream lengths are assumed to be infinite. A good example is an application that monitors and analyzes network traffic. A network may contain thousands of nodes that transmit data, which need to be analyzed in real time to detect critical conditions such as congestion. In traditional database systems, query optimization is fundamental to improving query performance. Similarly, DSMSs require query optimization (Viglas and Naughton, 2002) (Viglas et al., 2003).

1.1 Multiple-Query Optimization in DSMSs

Optimization techniques proposed for traditional relational databases cannot be directly applied to queries in data stream systems because of the differences between the two systems. To illustrate, one of the differences is that DSMSs deal with continuous queries, whereas traditional databases deal with one-time queries. Continuous queries are persistent, which means they are issued once and remain active for some time; on the other hand, traditional database queries are transient. Data stream systems need to keep producing updated results continuously, unlike traditional database systems that need to produce results only once.

Another difference is that data in traditional database systems are static and are stored in the system. Note that traditional database systems focus on generating precise answers computed by stable query plans. On the contrary, data streams are unbounded and cannot be stored in the systems. Therefore, some operations in DSMSs process tuples on the fly, while other operations need to store some data in the memory for a short period of time in order to process queries. Thus, queries in data stream systems require techniques that process memory-resident data. Moreover, query plans need to be re-optimized during run-time as stream arrival rates may fluctuate.

The above-mentioned points necessitate changing the optimization objective of data stream systems from traditional database systems. Previous research on DSMSs, such as in (Viglas et al., 2003), aim at optimizing the run-time output rate. Zhu et al. (2010) show that CPU and memory consumptions of a query plan need to be below the available CPU and memory resources of the system for the system to produce results at an optimal rate. To elaborate, if the CPU cost of the plan exceeds the available CPU resources, the plan will be unable to keep up with the incoming stream items; therefore, the system cannot produce results at the optimal rate. Similarly, DSMSs require that the memory does not overflow, which will negatively impact the system. Thus, the CPU and memory costs should be considered when generating a query plan since they are precious resources in the stream context. Such query plans are referred to as qualified plans (Zhu et al., 2010).

1.2 Query Adaptation Techniques

There are several query adaptation techniques in DSMSs. Below, some of the most commonly used techniques in these systems are described.

One of these adaptation techniques is called load shedding. Whenever the system resources cannot keep up with the incoming streams, load shedding can be used to reduce the workload in the system (Tatbul et al., 2003; Babcock et al., 2004; and Tatbul and Zdonik, 2006). The main idea of load shedding is to drop some of the tuples either randomly or based on certain criteria. Load shedding introduces the notion of approximation in the produced results. There are some applications where all the data the applications monitor are important; therefore, load shedding is not applicable to such applications.

Another technique is operator re-scheduling. In this technique, systems schedule query operators to minimize resource consumption. A scheduler of a system dynamically chooses the next operator based on the current workload; it has to be able to choose between different scheduling decisions based on the optimization goals (Carney et al., 2003; Babcock et al., 2003; Babcock et al., 2004). A good example of this technique is the eddy approach, which enables the reordering of operators in a query plan as it runs (Avnur and Hellerstein, 2000).

Another technique is intra-operator adaptation. When stream characteristics change, a query operator in this technique adaptively changes the way it handles streams. A good example of this kind of operator is the XJoin operator, which has three phases (Urhan and Franklin, 2000). In the first phase, the operator joins tuples that exist in memory, and it is called memory-to-memory phase. In case the memory portion of the input stream is full, the tuples are flushed to disk. In the second phase, the disk-to-memory phase, the operator joins tuples that are in memory with others that exist on disk. The final phase is the disk-to-disk phase, which joins tuples that are missed by the previous two phases.

Query plan re-optimization is another adaptation technique used in DSMSs. Based on the statistics of input streams, a query optimizer dynamically changes the current query plan, and reorganizes plan operators. In other words, the shape of the plan is changed to improve the plan efficiency (Park and Lee, 2009; Zhu et al., 2010; Park and Lee, 2012).

1.3 Problem Definition

Some research, as stated in (Zhu et al., 2010), uses query optimization strategies that are based on either mutli-joins (MJoins) or Binary Join trees (BJtrees) and, furthermore, they exploit positive correlations between CPU and memory. MJoin is a method that uses a single step to join all participant streams, whereas BJtree method takes multiple steps to join all participant streams. Note that MJoin systems reduce memory use, sometimes at the expense of additional CPU requirements and the BJtree approach does the opposite because it stores intermediate results. Zhu et al. (2010) proposed a technique in which they not only use either BJtree or MJoin, but also their combinations, which is called JTree. They also proposed a Two-Layer JTree-Finder algorithm to minimize memory and CPU resources of a query plan. In the first layer, they produce an optimal MJoin using Viglas et al. (2003) algorithm. If the optimal MJoin exceeds the constraints on the CPU, they produce an optimal BJtree using Kossmann and Stocker's (2000) algorithm. If the optimal BJtree exceeds the constraints on the memory, they move into the second layer, in which they modify either the optimal MJoin or BJtree to include a mixture of both operators in one plan. Thus, they are trading off memory for improving CPU computation and vice versa. They also discuss how to apply their algorithm to optimizing multiple queries by allocating fixed resources to each query and then using their algorithm.

However, they do not state how the resources should be allocated initially. Furthermore, and more importantly, their focus was on the single query instance and they indicate that further work is needed for the multi-query case along with their preliminary suggestions on how to handle the multi-query case. Moreover, they do not discuss how the sharing of subexpressions common to queries can be exploited to improve the efficiency and effectiveness of processing multiple queries. We aim to explore how single query optimization can be extended to move towards optimization of multiple queries.

The challenge in designing any continuous query system is to keep computation and the amount of storage required to satisfy multiple continuous queries below the available system resources. As Zhu et al. (2010) stated, trying to solve the problem by treating both CPU and memory costs as a single cost is hard to obtain. What complicates the design of such a singular cost function is that there is no clear relationship between CPU and memory resources because they are in some cases positively correlated, that is when one cost is reduced the other one is also reduced, while in other cases they have a negative correlation (Zhu et al., 2010). Finding optimal strategy for queries that minimizes a specific cost is NP-hard both in relational DBMSs (Sellis and Ghosh, 1990) and DSMSs (Zhu et al., 2010).

To elaborate, the inputs of the system are queries, total available CPU and memory. Queries are represented by join graphs wherein the vertices are input streams, each one with an arrival rate, and edges denote join predicates between input streams together with join selectivities, one per edge. The objective is to find a sequence of join operators that answers queries with available CPU and memory resources. Another goal we have is to find operations that can be shared among different queries to reduce the total CPU and memory consumption.

1.4 The Proposed Approach

The proposed approach in this thesis incorporates two different techniques that extend the (Zhu et al., 2010) algorithm, which was developed to optimize a single query for the optimization of multiple queries. First, qualified subplans are generated for queries by extending the plan search space to include BJtree operators, MJoin operators, and a mixture of both. Second, these subplans are used to produce a complete global execution plan for multiple queries by sharing the results of common join operations in a set of continuous queries; therefore, redundant operators are avoided. Each join operation that appears in the queries has a sharing degree that indicates in how many queries it appears and hence how much the operation can be reused by those queries. When forming plans, in addition to using the actual cost of a join operation, we also use an amortized effective cost (we refer to it as an effective cost) that is defined as the actual cost divided by its sharing degree. It is the effective costs that we use in determining which join operation is to be included in the global plan.

The global plan is created by optimizing queries one at a time. When a query is optimized, the proposed method checks the global plan for any join operations that can be used by the query. Operators, in the global plan formed thus far that can be used by the optimized query, will be incorporated into the query plan as the results already exist. Using this technique, some queries may use expensive operators when sharing their results makes it a good choice across all queries. We also propose a new method that finds the order of queries in which they need to be optimized. This is beneficial because finding the proper order of queries helps in reducing the total CPU and memory consumption of the global plan if more operators that can be shared across queries are chosen for the global plan.

As the algorithm in (Zhu et al., 2010) for a single query considers JTree, which may include both MJoins and BJtrees, and also is performed under the constraint on CPU and memory resources, we utilize it in our approach while adapting it in allocating initial resources to each query, ensuring that common subexpressions are shared among queries and are not recomputed, and calculating the costs of join operators when scheduling them.

1.5 Summary of Contributions

The main contributions of the proposed approach are as follows:

- This thesis introduces a method with the aim of finding a solution to the problem of generating a qualified global plan, which satisfies both CPU and memory resource constraints for multiple queries by adapting MJoin and BJtree techniques.
- The proposed approach generates a qualified global execution plan for multi-way join queries by sharing computations and memory of common join operations of multiple queries. Choosing join operations with smaller amortized costs forms the qualified global plan.
- We proposed three heuristic techniques that find the order of queries based on the similarity between queries, and the weight of queries. The order of queries is

determined based on their costs, namely heaviest/largest to lightest/smallest, lightest/smallest to heaviest/largest, and the shared predicate technique. The shared predicate technique is superior to the other two techniques as it generates more shared operations.

- The proposed approach efficiently pre-allocates CPU and memory resources to each query. The allocation of the resources ensures that when there is a shortage of one of the resources, the optimization of queries will take it into an account when forming the plan.

1.6 Assumptions

- This thesis concentrates on tree queries with equi-join to reduce search space. However, the problem we aim to address is still NP-hard and hence heuristics are used.
- Queries are expressed as join graphs. Each vertex represents a stream, which is marked by the name of the stream, and an arrival rate per unit of time. Moreover, each edge indicates a join predicate between two input streams, and is marked by the join selectivity. Selectivities change once a join is performed.
- In a DSMS, a monitoring component can collects statistics of input streams, such as input rate and join selectivity. When a query is first arriving at the system, the DSMS may predict join selectivities. As in most, if not all, work on query optimization in relational DBMSs and DSMSs, we assume that selectivities for a join of two streams are independent of a query. That is, the selectivity of joining

two data streams (relations in relational DBMSs) R and S is the same regardless of whether the join was issued by a query *q1* or a query *q2*.

- We assume static optimization. That is, we perform optimization for a given set of queries and their parameters and statistics of data streams and join selectivities. We do not address the issues of dynamic/adaptive optimization that addresses the problem of changes to the query and data stream changes to statistics, such as selectivities and arrival rates, as queries are processed. Some of the approaches to this problem, referred to as adaptive/dynamic optimization techniques, are discussed in Chapter 2.

1.7 Organization

The remainder of this thesis is organized as follows: In Chapter 2, background and related work are represented. Additionally, CPU and memory cost models are examined for positive and negative correlations of both system resources. The proposed approach is described in detail in Chapter 3, while the effectiveness of the proposed algorithm in this thesis is highlighted using some experiments in Chapter 4. Finally, Chapter 5 concludes this thesis.

CHAPTER 2 : BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Context and Environment

The work proposed in this thesis concentrates on streaming data that arrive continuously at a very high rate in a DSMS, such that the data are pushed into the system to be processed against a set of standing continuous queries that are registered in advance. The streams are unbounded sequences of data items, and each data item is associated with a timestamp. Furthermore, the data items may arrive in a burst or in equally spaced intervals. Examples of the burst fashion are network traffic streams and phone call records. On the other hand, an example of the second type is a pull-based system where the system contacts devices to retrieve data periodically. The focus of this thesis is on streams with a burst arrival pattern that arrive in either a centralized system or a distributed system. A centralized dynamic query optimization is used to generate global plans for multiple queries, as depicted in Figure 2-1- it is adapted from (Heinz et al., 2008). The query registration module is the component responsible for registering the continuous queries. The query executor executes the query plan generated by the optimizer. While continuous queries are evaluated, the runtime monitoring component gathers statistics of join operations in the current execution plan. Moreover, it collects and analyses statistics of input streams, such as input rate and join selectivity. When the effectiveness of the current execution plan declines because the parameter values used in generating the current optimal plan have changed, the query optimizer is invoked to generate a newly optimized plan, based on the current parameters. To illustrate, if the current execution plan starts consuming more CPU and/or memory resources than the total available system resources, the query optimizer is invoked. Finally, the system dynamically transfers the current plan to the new optimized plan using the plan migration component shown in Figure 2-1, which is another important research area in the DSMS field.

Queries that are registered against streams are frequently defined in terms of sliding windows because of the unbounded nature of streams. The purpose of the windowing is to limit the scope of the join operators into something of tractable size over the lifetime of a query. Window constraints are explained in greater detail in section 2.1.3.

To clarify the query types presented in this thesis, a scenario from a sensor network is considered. Let us assume that we have an organization that has a number of sensors to monitor the temperature and humidity at different locations throughout the rooms. A large organization may require thousands of such sensors. This scenario can be modeled as a system with two different types of streams, one for humidity sensors and the other for temperature sensors. The schema of each stream can be formed as follows: [locationID, value, timestamp], where locationID indicates the location of the sensor that generates the stream, value is the sensor reading, and timestamp identifies the time at which the data item is generated or entered into the system. There are two streams, one for temperature, stream T, and one for humidity, stream H. Queries, in this thesis, are SQL select statements where each query joins multiple streams. A query that continuously joins two streams, one generated from a temperature sensor and the other from a humidity sensor, within a 60-second interval can be specified as follows:

Query 1:

```
        Select *
        From Temperature T, Humidity H [Range 60 sec]
        Where T.locationID = H.locationID
```

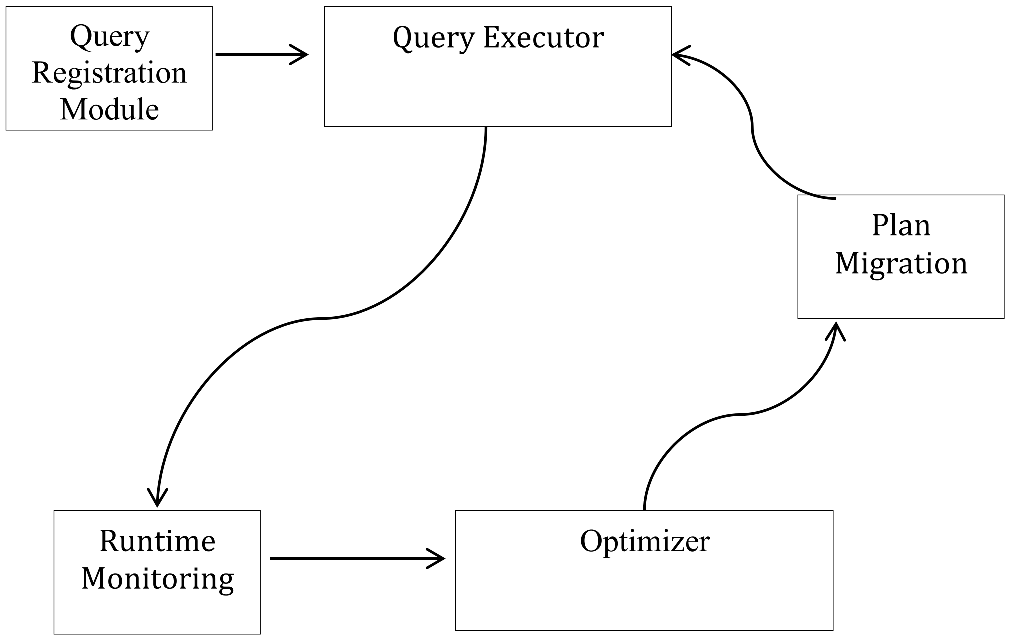


Figure 2-1 The overall system architecture adapted from (Heinz et al., 2008)

2.1.2 Streaming Applications

In this section, a number of data stream applications are represented not only to show the context of streaming data, but also to give an overview of query types that require the support of a DSMS.

2.1.2.1 Financial Tickers

Financial tickers contain applications that monitor tens of thousands of data streams, which feed from financial markets to help in making decisions. Moreover, these applications assist in finding correlations among all pairs of streams. An example of a query is to find stocks priced over $50 for the last 15 minutes.

2.1.2.2 Online Auction

Online auctions have been popular over the past decade. People from all over the world use them to trade goods. Online auction applications monitor prices and stream them to the users. An example query in an online auction is to find all auctions that closed within 60 minutes of their opening. Another example of a continuous query is to continuously monitor and compute the highest bids that occur in the last 20 minutes.

2.1.2.3 Sensor Networks

Sensor networks generate streams of data which can be used in monitoring applications. These applications monitor physical or environmental conditions, such as humidity, temperature, and pressure. An example of an application that monitors physical conditions is an application that deals with area monitoring. In area monitoring, a number of sensors are deployed over a specific area to trace human movement. Applications that monitor environmental conditions include, among others, forest fire detection, power consumption monitoring, and air pollution monitoring. An example of a query type included in sensor networks is a query that requires aggregating sensor data scattered over the network to monitor power consumption, and report it to a power station.

2.1.2.4 Transaction Log Analysis

Applications that analyze data stored in transaction logs of Intranets, web sites, and web search engines can provide valuable information for online searchers. Monitoring these transaction logs is important to achieve certain goals such as finding overloaded web servers. A query example of a web usage log is one that identifies overloaded web servers, which assists in rerouting users to backup servers.

2.1.3 Window Constraints

The size of the states in stateful operators can grow endlessly, and we need to have size constraints. Windows solve this problem. There are three types of window constraints: sliding windows, fixed windows, and landmark windows. Sliding windows are windows that have two moving endpoints (new items replace old ones). Fixed windows have two fixed endpoints, and landmark windows have fixed and moving endpoints. Sliding window constraints are used in this thesis. The definition of a window size has two forms. It can be defined in terms of a time interval, in which case they are called time-based windows. Windows belonging to the second form are called count-based windows, where windows are defined in terms of the number of tuples. Time-based window constraints are used in the queries represented in this thesis. In Query 1, the tuples that arrive from T and H are joined only if they are within 60 seconds of each other.

Window constraints can be of a global or a local kind (Hammad et al., 2002). The former is the most commonly used window constraint in which all input streams in one query have the same window constraint (Hammad et al., 2008). For example, a query $S_1 \bowtie S_2 \bowtie S_3$ with a global window constraint means that $W_{S1} = W_{S2} = W_{S3}$, where W_{Si} is the size of the window of stream *i* and it is specified in the query predicate. Our thesis uses queries with the global window constraints. The local window constraint, on the other hand, is when stream operators have different window constraints.

Window sizes can be different between queries that are registered in a DSMS. To process a join operation that is shared by multiple queries with different window sizes, there are three different approaches that can be used, namely the largest window only (LWO), the smallest window first (SWF), and a greedy approach based on the previous two approaches (Hammad et al., 2003). In the LWO approach, the result of the largest window is generated; then, the results of other windows are extracted from the largest window. In SWF, the results of windows are produced in ascending order of window size. In the final approach, windows are divided into groups based on their window sizes. Groups are evaluated in ascending order of the largest window size of each group. Thereafter, the LWO approach is used to produce the results of each group.

2.1.4 Correlation Between Resource Usages in Join Methods

The CPU and memory usage are always correlated with each other, and this correlation can be either positive or negative. In Figure 2-2, a box indicates an intermediate state where tuples are stored, and a circle indicates the CPU process of joining tuples. In the BJtree method depicted in Figure 2-2 (a), the resource usages are positively correlated. As the number of intermediate results increases, the system requires more CPU resources. Furthermore, they have a positive correlation when using the MJoin method, which is shown in Figure 2-2 (b). Increasing the number of stream tuples results in increasing both CPU and memory resources. On the contrary, the JTree method utilizes the negative correlation between the CPU and memory usages. Adding intermediate states to avoid recomputation can reduce CPU consumptions. As shown in Figure 2-2 (c), JTree is composed of a combination of BJtree and MJoin operators. A more detailed explanation may be found in (Zhu et al., 2010).

This thesis utilizes the positive and negative correlations between the resources usages to generate a qualified global plan that answers multiple queries. Recall that a qualified plan requires less CPU and memory resources than the total available system resources.

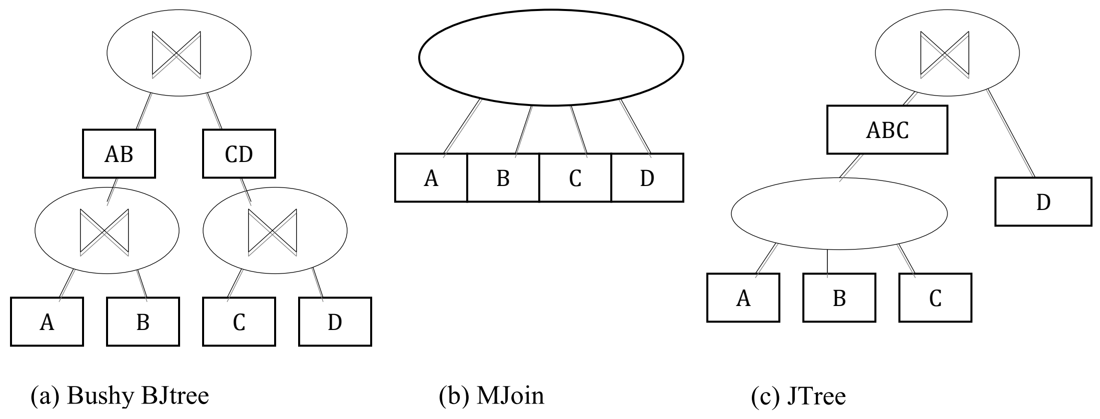


Figure 2-2 BJtree, MJoin, and JTree

2.1.5 Cost Models

A global query plan can be represented by a graph that contains AND-nodes, OR-nodes, and a set of edges (Park and Lee, 2012). The join operation is executed by the AND-node, whereas the result of the join operation is maintained in the OR-node. A qualified global plan *P* is composed of binary joins, MJoins, or a mixture of both, the result of which produce an answer for each query in *Q*. Queries in *Q* are continuous queries with equi-join predicates and each query is an SQL select statement where each query chooses from different source streams. In other words, each query q_i in *Q* is restricted to a conjunction of simple terms that are restricted to equality, i.e., where each term is of the form ($r.A = s.A$), where r and s are streams that contain a joining attribute A. Moreover, the size of windows between queries may differ, but the size of windows in one query is the same for each stream referred to by the query. This type of query is called a uniform clique window join, where all the streams in each query have the same window size (Hammad et al., 2008).

The global plan P produces the results of all the queries included in Q. A JTree plan is represented by a graph as shown in Figure 2-3 (a). The graph contains a set of nodes N and an acyclic set of edges E. A part of a global execution plan for multiple queries is referred to as a subplan as shown in Figure 2-3 (b) (Park and Lee, 2012). The nodes in the subplan are connected, i.e., the graph is connected. An AND-node is shown as a circle, whereas a box indicates an OR-node.

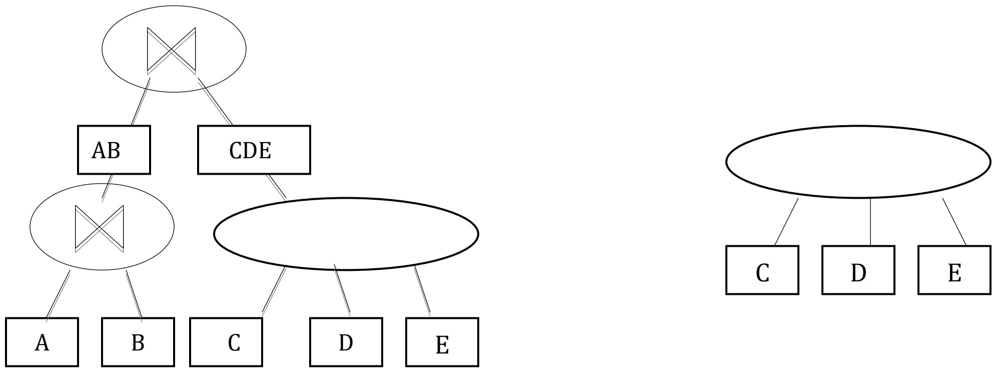
A global execution plan and all of its subplans for a set of queries have the following properties that are adopted from Park and Lee (2012). The number of incoming edges, and the number of outgoing edges of a specific node (n_i) where $n_i \in N$ are defined as follows:

Incoming edges of (n_i) ≥ 2 if $n_i \in$ AND-nodes
Incoming edges of (n_i) $= 0$ if $n_i \in$ OR-nodes, where n_i is a base stream
Incoming edges of (n_i) $= 1$ otherwise

Outgoing edges of (n_i) $= 1$ if $n_i \in$ AND-nodes
Outgoing edges of (n_i) ≥ 0 if $n_i \in$ OR-nodes

Note that the number of outgoing edges of an AND-node is always equal to 1 because it is only fed to an OR-node. This OR-node is a standard representation to hold the results.

In other words, an OR node is a results buffer. Inserting and deleting tuples from OR-nodes require some CPU utilization.



(a) An example of a global plan (b) A subplan

Figure 2-3 A global plan and its subplan

Table 2-1 Terms in the cost models (Zhu et al., 2010):

Term	Meaning
C_i	Cost of inserting a tuple (ms)
C_d	Cost of deleting a tuple (ms)
C_j	Cost of joining two tuples (ms)
λ_x	Input rate of stream x (tuples/sec)
σ_{xy}	Join selectivity of a join operation on operand streams x and y
W_x	Window size constraint on stream x.
$\|S_x\|$	Number of tuples in state x, where $\|S_x\| = \lambda_x W_x$

The total cost of a global execution plan for a number of queries is the sum of the costs of all the join operations in the plan. The join operation is executed by the AND-node, whereas the result of the join operation is maintained in the OR-node. The cost of maintaining tuples in an OR-node can either be the number of tuples arrived from an input data stream in a time unit or the number of result tuples produced by the predecessor AND-node in a time unit. The cost of forwarding the tuple to the subsequent operation (if any) is zero. In general, a symmetric stream hash join (Viglas et al., 2003) with a unit-time-based cost model (Kang et al., 2003) is employed. Table 2.1 describes the terms used in the cost model. The cost of the symmetric stream hash model is explained in sections 2.1.5.1, 2.1.5.2, and 2.1.5.3.

An AND-node may have two or more operand streams. Moreover, it continuously receives tuples from the input streams. Whenever new tuples are received from the operand streams, they are joined. The evaluation cost of an AND-node is the sum of the costs of joining the newly arriving tuples of all participated streams. The cost of an AND-node is the CPU processing of joining tuples, which will be explained in detail later in this chapter.

An OR-node is a standard representation to hold the results. The cost of an OR-node is the number of the result tuples that exist in the node. In other words, the cost of an OR-node is the memory cost, which will be clarified in the following sub-sections. However, inserting and deleting tuples from OR-nodes of the operand streams are considered when calculating the CPU cost.

The total CPU costs of a global execution plan P is the total cost of all AND-nodes in addition to inserting and deleting tuples from OR-nodes, whereas the total cost of the memory in the global plan is the total size of OR-nodes in the plan.

The following subsections show the CPU and memory costs for BJtree and MJoin plans. The CPU cost is the time the CPU needs to process all newly arrived tuples in a time unit, whereas the memory cost is the size of all tuples the plan needs. To describe the procedure of calculating CPU and memory costs, the example query represented as a join graph in Figure 2-4(a) is used. Each vertex represents a stream, which is marked by the name of the stream, and an arrival rate per unit of time. Moreover, each edge indicates a join predicate between two input streams, and is marked by the join selectivity. Given this join graph, Figure 2-4 (b) shows the optimal plan for MJoin processing as determined by the algorithm in (Viglas et al., 2003), whereas Figure 2-4 (c) depicts the BJtree plan produced by the algorithm introduced in (Kossmann and Stocker, 2000).

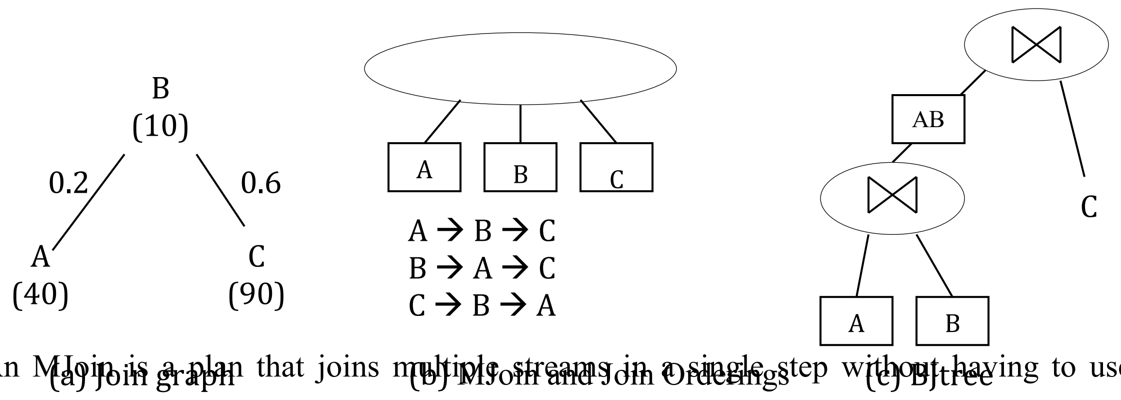


Figure 2-4 Example query and Two Joins

An MJoin is a plan that joins multiple streams in a single step without having to use multiple stages as in the case of a BJtree plan. Each input stream in the MJoin plan is joined with other streams in a particular order, which can be determined by the MJoin algorithm (Viglas et al., 2003). Subsections 2.1.5.1 and 2.1.5.2 show a summary of the frequently-used cost analysis of BJtree and MJoin, respectively, as presented in (Zhu et al., 2010).

2.1.5.1 Cost Analysis for BJtree

The CPU cost of a BJtree operation is the total cost of processing newly arrived tuples from each input stream. For example, we have three input streams *A*, *B*, and *C* that we need to join. A BJtree plan is shown in Figure 2-4 (c). When new tuples arrive from *A*, first they are inserted into state *A*, at the cost C_i. The cost of inserting tuples from input stream *A* into state *A* is $\lambda_A * C_i$. Moreover, tuples that are outside the time-frame (W) are deleted from state *A* at the cost C_d, The number of tuples that need to be deleted is equivalent to λ_A and that is computed as follows: $\lambda_A * C_d$. Then, the new tuples are joined with the tuples that are in state *B* at the cost C_j, and the joined tuples are added into intermediate state *AB*. Furthermore, the old tuples are deleted from state *AB*, and the cost model used to compute this step is $\lambda_A * \lambda_B * \sigma_{AB} * W * (C_j + C_i + C_d)$. In the following step, the tuple from state *AB* is joined with the tuples that exist in state *C*, and the computation of this step is $\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj$. There is no need to store the produced results, since there is no state |ABC|. In other words, there are no inserting and deleting operations in the last step. The CPU cost for stream A in a unit time is $CPU_A = (\lambda_A * C_i) + (\lambda_A * C_d) + (\lambda_A * \lambda_B * \sigma_{AB} * W * (C_j + C_i + C_d)) + (\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj)$. New tuples that arrive from input *B* follow the same steps, whereas tuples from *C* are, first, inserted into state *C*. Subsequently, they join directly the tuples in state *AB*. The CPU cost for *C* in a unit time is $CPU_C = (\lambda_C * C_i) + (\lambda_C * C_d) + (\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj)$.

The following is the total CPU cost for joining (A ⋈ B) ⋈ C.

$$CPU_{bjtree} = (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) + (3 * \lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj) + (2 * \lambda_A * \lambda_B * \sigma_{AB} * W * (C_j + C_i + C_d)) \quad (2\text{-}1)$$

The memory size is as follows:

$$\text{MEMORY}_{\text{bjtree}} = |\text{State A}| + |\text{State B}| + |\text{State C}| + |\text{State AB}| = (\lambda_A * W) + (\lambda_B * W) + (\lambda_C * W) + (\lambda_A * \lambda_B * \sigma_{AB} W^2) \qquad (2\text{-}2)$$

The join order affects the size of the intermediate state. Choosing a better join ordering decreases the memory size. If memory size is decreased, the CPU cost is reduced as well as they are positively correlated.

2.1.5.2 Cost Analysis for MJoin

The CPU cost of the MJoin operation consists of the total cost of processing newly arrived tuples from every input stream. For example, *A*, *B* and *C* are the streams we want to join using an MJoin plan. The join ordering of each stream is shown in Figure 2-4 (b). When new tuples arrive from *A*, they are inserted into state *A*. The tuples that are outside the window size are deleted. The new tuples are then joined with the tuples in state *B*, and the resulting tuples are joined with the tuples in state *C*. The CPU cost for *A* in a unit time is $\text{CPU}_A = (\lambda_A * C_i) + (\lambda_A * C_d) + (\lambda_A * \lambda_B * \sigma_{AB} * W * C_j) + (\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj)$. The same process is applied to the new tuples from *B* and *C*. The following is the total CPU costs for MJoin.

$$\text{CPU}_{\text{mjoin}} = (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) + (3 * \lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * Cj) + (2 * \lambda_A * \lambda_B * \sigma_{AB} * W * C_j) + (\lambda_B * \lambda_C * \sigma_{BC} * W * Cj) \qquad (2\text{-}3)$$

The memory size of the MJoin is as follows:

$$\text{MEMORY}_{\text{mjoin}} = |\text{State A}| + |\text{State B}| + |\text{State C}| = (\lambda_A * W) + (\lambda_B * W) + (\lambda_C * W) \qquad (2\text{-}4)$$

The optimal join orderings of input streams decreases the total CPU cost of an MJoin plan (Viglas et al., 2003). Figure 2-4 (b) shows the optimal join orderings in our example.

2.1.5.3 Cost Analysis for JTree

The CPU cost of a JTree method is the total cost of processing newly arrived tuples from each input stream. The JTree method is a combination of BJtree and MJoin operators; thus, the CPU costs of a JTree method is calculated by combining equations (2-1) and (2-3). Since we can not generate a JTree plan from the example shown in Figure 2-4, let us consider the JTree plan shown in Figure 2-2 (c), which is joining ($A \bowtie B \bowtie C$) $\bowtie D$. The reason is that the JTree plan needs at least 4 input streams to be formed.

First, input tuples that arrive from streams A, B, and C are joined together using equation (2-3). The join ordering of each stream must be known a priori to calculate the CPU costs of an MJoin operator. Since the resulting tuples need to be stored in an intermediate state, the cost of inserting and deleting tuples from the new intermediate state needs to be considered when calculating the total CPU costs. Assume that the join ordering of each stream is the one shown in Figure 2-4 (b). The CPU cost for A in a unit time is $CPU_A = (\lambda_A * C_i) + (\lambda_A * C_d) + (\lambda_A * \lambda_B * \sigma_{AB} * W * C_j) + (\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * (C_j + C_i + C_d))$. The same process is applied to the new tuples from B and C. The resulting tuples are stored in state ABC. Tuples from D are first inserted into state D and then they join directly the tuples in state ABC. The CPU cost for D in a unit time is $CPU_D = (\lambda_D * C_i) + (\lambda_D * C_d) + (\lambda_A * \lambda_B * \lambda_C * \lambda_D * \sigma_{AB} * \sigma_{BC} * \sigma_{CD} * W^3 * Cj)$. The following is the total CPU costs for JTree.

$$CPU_{Jtree} = (\lambda_A + \lambda_B + \lambda_C + \lambda_D)(C_i + C_d) + (4 * \lambda_A * \lambda_B * \lambda_C * \lambda_D * \sigma_{AB} * \sigma_{BC} * \sigma_{CD} * W^3 * Cj) + (3 * \lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^2 * (C_j + C_i + C_d)) + (2 * \lambda_A * \lambda_B * \sigma_{AB} * W * C_j) + (\lambda_B * \lambda_C * \sigma_{BC} * W * Cj) \qquad (2\text{-}5)$$

The memory size is as follows:

$$MEMORY_{Jtree} = |\text{State A}| + |\text{State B}| + |\text{State C}| + |\text{State D}| + |\text{State ABC}| = (\lambda_A * W) + (\lambda_B * W) + (\lambda_C * W) + (\lambda_D * W) + (\lambda_A * \lambda_B * \lambda_C * \sigma_{AB} * \sigma_{BC} * W^3) \qquad (2\text{-}6)$$

2.2 Related Work

The explosive growth of sensors and RFID networks and utilizing them in real-time scenarios caused a dramatic increase in streaming data on the Internet (Sirish et al., 2003). Therefore, there was an urgent need for systems that can process this kind of data since streaming data are different in nature from the data in traditional databases. There are two fundamental differences between data stream management systems and traditional database systems. First, streaming data is infinite, whereas data is finite in traditional systems. Streaming data does not need to be stored entirely on the systems in contrast to data in traditional systems. The second difference is that queries in DSMSs are persistent, which means that queries are issued once and they remain active in the system for a long time. On the other hand, queries in DBMSs are transient. They return results and then they become inactive. This emergence of streaming data has resulted in the development of streaming database systems (Chen et al., 2000; Rundensteiner et al., 2004; Abadi et al., 2005).

2.2.1 Join Methods

Join has been extensively studied in static databases since it is one of the most commonly used operations in database systems. Viglas (2005) predicts that join will be one of the most intensive research areas in the stream context.

The two most common methods used to execute multi-join queries over streams are MJoin (Viglas et al.; 2003) and BJtree (Kossmann and Stocker, 2000). Both techniques mostly use hash-based join algorithms, which were primarily introduced by DeWitt (1984) for static databases. Wilschut and Apers (1990) introduce the symmetric hash join, which deals with the streaming nature. Urhan and Franklin (2000) provide a technique to overcome memory overflow by introducing an efficient way to spill overflowing tuples to disk and join them later to produce the output. Urhan and Franklin (2000) improve the performance of the symmetric hash join algorithm proposed by Wilschut and Apers (1990), and enhance the response time in stream contexts. The primary reason for using a hash-based join implementation is that previous works have proven, in most cases, its superiority over other implementation such as nested-loop join (Zhu et al., 2010). Each join operator in a hash-based join keeps one state per input stream to store tuples for future joins.

To process continuous queries with multiple joins, the works before (Viglas et al., 2003) use the BJtree method, which contains binary join operations that store intermediate results. This method requires multiple steps to join all participant streams. Since the BJtree method stores intermediate results, it requires memory space for the storage. There are two different types of BJtrees: linear BJtree and bushy BJtree. The inputs of a bushy tree are intermediate results, except the leaves, which are all base streams. On the other hand, the inputs of each join operator in linear trees are intermediate result and a base stream. However, this is not the case for the first operator in linear trees, which joins tuples received from base streams. For the purpose of reducing the time complexity, some query optimizers restrict the search space to linear BJtree, which may reduce the chance of finding a good query plan (Selinger et al., 1979). In fact, prior works have shown that a bushy BJtree in many cases has better performance than its counterpart (Viglas and Naughton, 2002).

MJoin is the second common method used to perform joins in stream environments (Viglas et al., 2003). Instead of requiring multiple steps to perform the join, MJoin uses a single step to join all participant streams. When a new tuple arrives, it is joined with the other streams in a specific order. One of the differences between BJtree and MJoin is that MJoin does not store intermediate results as in the case of BJtree. Therefore, MJoin requires less memory compared to the BJtree method; however, this is performed at the expense of increasing the CPU costs. This problem of excessive recomputation of intermediate results in MJoin was acknowledged earlier by Viglas and his colleagues (2003) when inventing the MJoin method. However, Viglas and his colleagues (2003) do not provide a solution to tackle this problem. They suggest that a large MJoin, which has more than four participant streams, can be split into two MJoin operators; however, they do not account for memory constrains/costs.

2.2.2 Processing Multiple Queries

Similar queries can be executed by sharing the computation of query plans (Chen et al., 2000). Madden et al. (2002) suggest indexing query predicates for the purpose of processing multiple queries.

In the first approach, similar queries share the same computation, which can be performed by building a common query plan (Chen et al., 2000). To illustrate, continuous queries are divided into groups based on the similarity between the queries, and each query belonging to a group shares a common subplan with other queries in this group. The query plan contains all operations that answer all queries. This approach is beneficial in reducing redundant operations when dealing with multiple queries. Although some works have been accomplished using this technique of sharing query plans (Chen et al.,

2000; Park and Lee, 2012) to reduce memory or CPU cost, to the best of our knowledge no one has tried to address the problem of generating qualified plans for multiple queries, which is one of objectives of this thesis. Roy et al. (2000) emphasize the significance of integrating a query plan with sharing and materializing common results. However, Roy et al. (2000) do not consider pipelining common results to their uses without materializing them. Dalvi et al. (2003) suggest pipelining common results without materializing them to reduce memory usage. Not all common results can be pipelined. Dalvi et al. (2003) do not take into account CPU processing time. In DSMSs, query plans need to be generated as quickly as possible. Thus, the work in (Dalvi et al., 2003) is not applicable to DSMSs since it requires considerable time to generate a plan. Park and Lee (2012) improve the technique introduced by Dalvi et al. (2003). They make it applicable to the data stream environment. Park and Lee's algorithm traces a set of promising subplans when generating a global plan for multiple queries. The BJtree method is used in their technique to generate common subplans. The work by Park and Lee (2012) does not consider both CPU and memory consumption and the correlation between them.
The second approach is to index query predicates and store them in a table. The attribute values of each newly arrived tuple are extracted and matched against the available query index to determine which queries are satisfied by the tuple (Golab and Özsu, 2003).

2.2.3 Overview of Other Related Work

The previous research dose not tackle the problem of generating qualified global plans for multiple queries. Most previous works utilize the positive correlation between CPU and memory usages. The works that focus on optimizing BJtree, such as in (Babu et al., 2004; Urhan and Franklin, 2000), try to minimize the memory usage by reducing the size of intermediate results, and assume that this will reduce the CPU consumption. The works that focus on enhancing the MJoin method, such as in (Viglas et al., 2003, Madden et al., 2002), aim to reduce the CPU costs needed to recompute intermediate tuples without storing any intermediate results.

There are two additional works that utilize the positive and negative correlations between CPU and memory usages. The first work is the Adaptive Caching algorithm (Babu et al., 2005), which optimizes join performance by considering BJtree plans, MJoin, and the entire spectrum between them. To elaborate, the Adaptive Caching technique starts with a single MJoin operator and then places/removes caches for specific intermediate results. There are differences between the technique proposed in this thesis and the Adaptive Caching technique. First, the Adaptive Caching technique does not consider solutions with overlapping candidate caches. It restricts the problem space to a set of nonoverlapping caches because considering overlapping caches will complicate cache management and optimization. To illustrate, nonoverlapping caches are the ones that do not have join operators in common. Second, the Adaptive Caching technique looks at the operator-level. Third, the Adaptive Caching technique does not consider sharing the results of common join operations between queries.

The other similar work is the Two-Layer JTree-Finder algorithm (Zhu et al., 2010), which finds a qualified query plan by extending the solution space to include BJtree, MJoin, and a mixture of both. The first layer generates, first, an MJoin plan. If the plan requires more CPU resources than the total available resources, the first layer generates a BJtree plan. If the plan is still not qualified, the second layer is triggered. In the second layer, a mixture of MJoin and BJtree subplans is generated to find a global plan that is adherent to both CPU and memory resources. In case a qualified plan is not found, other techniques, such as load shedding (Tatbul et al., 2003) and memory spilling (Urhan and Franklin, 2000), can be used. The work in (Zhu et al., 2010) deals only with a single query and has not yet been extended to handle multiple queries.

Regarding some of the recent works in the field, Heinze et al. (2014) set latency as the optimization target for multiple queries in a distributed data stream system. The optimization component of their proposed system finds reusable operators in different queries to construct a global plan. Lohman (2014) discusses some issues that have not been solved yet in query optimization, one of which is the problem of redundant predicates that may appear in queries. He gives an example of two queries that differ in only one predicate, such that the predicate is redundant, that is, it is implied/subsumed by the other predicates. Chen et al. (2013) focus on optimizing multiple top-k-queries over uncertain data streams, where data is incomplete, imprecise, and misleading. Techniques that are used in multi-query optimization over data streams cannot be applied to multiple top-k query processing over uncertain data streams because of the complex semantics of uncertainty (Chen et al., 2013). To et al. (2017) focus mainly on managing states in data stream systems in which they provide some solutions to overcome the issue of having scarce memory resources. One solution is to not restrict the processing to local memories. Large states can be stored in a distributed system for joining multiple data streams to ensure high scalability. Arasu et al. (2016) give insight into their data stream management system. In their system, they use load shedding when system resources overflow. They ignore the idea of producing accurate results, which is one focus of this thesis.

CHAPTER 3 : GLOBAL PLAN GENERATION ALGORITHM

One obvious way to "optimize" multiple queries is to optimize each query individually and then check which operations across queries can be shared. Our proposed method is heuristic and it has the following contributions:

- Determine the order in which queries should be optimized.
- Pre-allocate CPU and memory resources to each query to ensure that when there is a shortage of one of the resources, the optimization of queries will take it into an account when forming the plan.
- Consider sharing computations and memory of common join operations when generating a global plan.
- Satisfy both CPU and memory resource constraints by considering MJoin operators, Bjtree operators, and a mixture of both when forming the plan.

Our primary objective of generating a global plan is to answer multiple queries while minimizing resource usage by using both BJtree and MJoin operators and sharing the results of common join operations. Generating a query plan only using one type of operators, whether they are BJtree or MJoin operators, can miss the opportunity to generate a plan that adheres to both CPU and memory usage. Moreover, when dealing with multiple queries, sharing computation is significant to eliminating redundant work.

Our method creates a global plan by optimizing individual queries and adding their operators to the global plan: when optimizing a query, the proposed method checks the global plan for any join operations that can be reused or shared by the query. In case there are sharable global plan join operators (i.e., operators of which results can be shared by more than one query) that can be used by the query being optimized, they may be used even if they are expensive for the query being optimized. Using this technique, one combination of joins may be more expensive for any single query but the amortized cost across a number of queries can make sharing to be a good choice across all queries. Because our method optimizes each query individually, the proposed method needs to determine the order of queries in which they need to be optimized. We propose three techniques to find the order of queries and they are described in the following section. Our method also allocates a fixed amount of the respective CPU and Memory resources for each query based on the number of input streams, input rates, and join selectivity of join operations. A global execution plan is generated by optimizing queries individually, while searching for shared operators when creating a plan for an individual query, and incorporating their plans into the global plan. Subsequently, the global plan will have all subplans that are required to evaluate the set of queries. To increase the likelihood of finding common join operations, the effective cost of a join operation is used in this process. The effective cost of any join operation takes into account the number of queries that can reuse the operation's result.

In Section 3.1 we present how the order of queries used by the global plan generation algorithm is determined. How resources are initially allocated to each query is described in Section 3.2. Determining which join operations can be shared by multiple queries is described in Section 3.3. Section 3.4 describes the global plan generation algorithm without details on the generation of qualified sub-plans that are then described in Section 3.5.

3.1 The Order of Queries

Our method needs to determine the order of queries before optimizing them. To order the queries, our method exploits the similarity between queries and the weight of queries. Each join predicate has a degree of similarity *d* (*j,Q*) that represents the number of shares of the join *j* amongst the queries in *Q*. To determine *d* (*j,Q*) for a given predicate, we simply examine the processed queries and count how many of them have the predicate and thus determine the degree of similarity *d* (*j,Q*). Intuitively, for a given predicate (*r.A* = *s.B*), where *r* and *s* are relations and *r.A* and *s.B* are the joining attributes, if *d* (*j,Q*) = 3, then that predicate appears in three queries. If we perform the join using this predicate, then the result of this join is shared by three queries – which is preferred to executing the same join in evaluating each query individually. Thus, the higher the degree of similarity, the better it is to schedule the join so that its result can be shared. However, the cost of executing a join itself is also relevant. Higher the cost of the join itself, then its sharing needs to be higher for it to be selected. We represent this property by the weight-measure of a join predicate that is proportional to the cost of a join, which depends on the number of tuples in the window that need to be joined, and hence on the arrival rates and the number of tuples formed by the join, which depends on the predicate's selectivity. Thus the gaged weight of a join predicate is calculated by equation 3.1, which multiplies the join selectivity with the arrival rates of the operand streams *a* and *b*.

$$\textit{The gaged weight of a join predicate} = \lambda_a * \lambda_b * \sigma_{a.b} \qquad (3-1)$$

In short, the heavier the weight, the more expensive the processing of the predicate. Later in this Chapter, we use the term edge weight and it is calculated using equation 3.1.

The primary objective of finding the order of queries is to reduce the total CPU and memory consumption of the global plan. In other words, processing queries in different orders has an impact on generation of the global plan. We explore three heuristic techniques to find the order of queries based on their weights: heaviest/largest to lightest/smallest, lightest/smallest to heaviest/largest, and the shared predicate technique.

The ammortized weight of a binary join predicate j between streams a and b with respect to queries in set Q is : $w(j, Q) = \frac{\lambda_a * \lambda_b * \sigma_{a,b}}{d(j, Q)}$ (3 − 2)

The amortized weight of a join predicate, or from now on simply referred to as the weight of a predicate, is determined by equation 3.2, which divides the weight by the degree of similarity *d(j,Q)* that represents the number of shares of the join j amongst the queries in Q.

Let $Q = \{q_1, q_2, q_3, \ldots q_n\}$ be a set of queries, where each query is represented by an acyclic join graph. Query $q_i = (V_i, E_i)$ is comprised of a set of streams $V_i \sqsubseteq V$ and edges E_i as predicates (joins) between streams. Let the join selectivity of edge $e_j = (K, L)$ be $\sigma_{K,L}$,. Moreover, the number of edges / predicates in q_i is $|E_i|$.

Let $\sum_{x \in S}$ denote the sum over all x in S. The weight of a query q_i, relative to query set Q, is the summation of the weight of all join predicates that exist in the query and it is calculated as follows:

$$w(q_i, Q) = \sum_{(K,L) \in E_i} \frac{\lambda_K \cdot \lambda_L \cdot \sigma_{K,L}}{d((K,L), Q)} \qquad (3-3)$$

Sharing of results amongst the queries is encouraged by defining the cost of a predicate to be its weight divided by the degree of sharing. In essence, we explicitly amortize the cost of the join across all sharing in the query, which reduces the per-query cost of the operation. Thus, when we consider the cost of any single query, the use of the shared operator may become a low-cost option for its query plan, which means the join will likely be selected.

The three algorithms, which determine the order of queries for the global plan generation algorithm, are now described. Using equations (3-2) and (3-3), the algorithms calculate the weight of a predicate appearing in a query and the weight of a query.

3.1.1 Heaviest to Lightest:

In this technique, queries are ordered from the one with the heaviest weight to the one with lightest weight. Therefore, the heaviest queries will be processed first. As queries are individually optimized, CPU and memory resources are allocated to their plans. This method will ensure that queries requiring "heavy" resources are optimized first and thus it is expected that there are resources available to process them.

One drawback of this approach is that it does not encourage generating shared operators at an early stage. In a later stage, if there is a shortage in one of the system resources such as memory, the system will prioritize operators with low memory consumption over operators that can be shared by multiple queries. Therefore, this technique, in some cases, does not fully exploit the similarity between queries. The goal here is to generate join operations that can be shared between queries as soon as possible, so CPU and memory resources of the shared operations are shared by the rest of the queries if possible. This is significant when considering the pre-allocation of resources to queries, which will be discussed in section 3.2. Algorithm 1 shows how to order queries using this approach. Keep in mind that the degree of similarity that represents the number of shares of the join amongst the queries may change when a query is removed. This happens when there are predicates shared with the removed query. Thus, after removing a query in line 8 of Algorithm 1, the weights for queries need to be recalculated, and that is the reason why we keep recalculating the weights for queries in each iteration of algorithm 1 (lines 3-5).

Algorithm 1 Finding_Order_of_Queries (Heaviest to Lightest)

Input: queries $Q = \{ q_1, q_2, \ldots, q_n \}$, where each query is a Join Graph

Output: Order of queries in an Ordered_list.

```
1: Ordered_list = empty;
2: While ($Q$ != null);
3:         For each query $q \in Q$ do
4:                 calculate and store query weight w($q$,$Q$) using equation (3-3)
5:         End for
6:         let $q_{largest\_index} \in Q$ be the query with largest weight w($q_{largest\_index}$,$Q$)
7:         append $q_{largest\_index}$ to Ordered_list;
8:         remove $q_{largest\_index}$ from $Q$;
9: End while
10: Return Ordered_list
```

3.1.2 Lightest to Heaviest:

Unlike the first approach, queries in this technique are ordered from the one with the lightest weight to the one with the heaviest weight. This approach has a chance to generate operators that can be shared across queries at an early stage because it exploits the similarity between queries to some degree. However, it may miss some opportunities for generating operators that can be shared among queries. Figure 3-1 illustrates how some queries may ignore sharing operators. Let us assume we have sufficient system resources. There are three queries: query 1 has the lightest weight, query 2 has the second lightest weight, and query 3 has the heaviest weight, which is calculated by equation 3-3. In this example, when processing query 1, B and C are joined first because they have the lightest weight. Then, their result is joined with stream A. If we process query 2 first, A and B are joined first because their weight is smaller than joining B and D. Thus, processing query 2 first is more beneficial because it generates a join operation that can be shared between two queries.

Queries
1) A ⋈B⋈ C
2) A ⋈B⋈ D
3) A ⋈C⋈ D

Input rate for streams
A = 10
B = 12
C = 10
D = 30

Selectivity factor
A ⋈ B = 0.1
B ⋈ C = 0.01
B ⋈ D = 0.2
A ⋈ C = 0.9
C ⋈ D = 0.5

Figure 3-1 Example of ordering queries from lightest to heaviest

It is sometimes beneficial to process a heavier query at an early stage if its results can be shared by other queries as shown in the example of Figure 3-1. Optimizing all heavy queries as last may mean that resources may not be available for them. For instance, generating query plans for all but the last query may not leave sufficient memory (CPU) available for the last query, which may require a lot of memory (CPU) resources and thus the global qualified plan may not be found. If that heavy query was optimized early, a global plan may have been found. Algorithm 2 shows how to order queries using this technique.

Algorithm 2 Finding_Order_of_Queries (Lightest to Heaviest)

Input: queries $Q = \{ q_1, q_2, \ldots, q_n \}$, where each query is a Join Graph

Output: Order of queries in an Ordered_list.

```
1: Ordered_list = empty;
2: While ($Q$ != null);
3:     For each query $q \in Q$ do
4:         calculate and store query weight w($q,Q$) using equation (3-3)
5:     End for
6:     let $q_{smallest\_index} \in Q$ be the query with smallest weight w($q_{smallest\_index}$,$Q$)
7:     append $q_{smallest\_index}$ to Ordered_list:
8:     remove $q_{smallest\_index}$ from $Q$;
9: End while
10: Return Ordered_list
```

3.1.3 Shared predicate technique:

In the previous two algorithms the order of queries was determined by properties of queries, one ordering the queries from lightest to heaviest, while the other from heaviest to lightest, where the weight of query is determined using the equation (3-3). This technique is used to solve the issue discussed in section 3.1.2 by selecting the order of queries not on the properties of queries, but rather on properties of predicates that queries contain and favoring those queries that contain predicates that are shared across queries. The method examines the join predicates that are shared between at least two queries and chooses the one with the smallest weight. Recall that the weight of a join predicate is calculated by dividing its weight by the degree of similarity. The heaviest query that shares this predicate is then placed into the order of queries to be optimized. We prioritize heaviest queries here because a poor query plan for heavy queries exhausts system resources more than any other types of queries. Some operations that exist in the global plan may force heavy queries to have poor query plan overall. Thus, processing heavy queries first reduces the probability of using unwanted operations that already exist in the global plan by those queries. After placing q_1 in the order, q_1 is removed from the set of queries and the weights of the remaining queries are re-calculated using equation 3-3. The process then repeats, the cheapest join predicate that is shared between multiple queries is selected and the heaviest query that shares it is set to be the second query in the order. This process will continue until all queries are placed in order. In case there is more than one predicate with the same weight, the heaviest query that shares any one of these predicates is chosen first. In case there does not exist shared predicate anymore, the smallest predicate is selected and the largest query that has it is placed next. The experimental results in section 4.2 show that this technique is superior to the other two as it generates more shared operations. Algorithm 3 shows how to order queries using this approach. Note that a shared join predicate, sometimes referred to as a shared predicate, is a predicate that is shared between at least two queries.

Algorithm 3 Finding_Order_of_Queries (Shared-predicate)

Input: queries $Q = \{ q_1, q_2, \ldots, q_n \}$, where each query is a Join Graph

Output: Order of queries in an Ordered_list.

```
1: Ordered_list = empty;
2: While (Q != null)
3:     For each query $q \in Q$ do
4:         For each join predicate j in each query q do
5:             calculate and store join weight w(j,Q) using equation (3-2)
6:         End for
7:         calculate and store query weight w(q,Q) using equation (3-3)
8:     End for
9:     If there is shared predicate, that is shared by different queries, still exists in queries $q \in Q$ do
10:        Find that predicate j, over all queries $q \in Q$, such that:
               1) j has the smallest weight w(j,Q) and
               2) j is shared by at least two queries from what left
11:        Find that query $q \in Q$, such that:
               1) q has the predicate j and
               2) weight w(q,Q) is the largest of all queries that share the predicate j
12:    Else // if there is no shared predicate anymore in queries $q \in Q$
13:        Find that predicate j, over all queries $q \in Q$, such that:
               j has the smallest weight w(j,Q)
14:        Find that query $q \in Q$, such that:
               1) q has the predicate j and
               2) weight w(q,Q) is the largest of all queries
15:    End If
16:    append q to Ordered_list:
17:    remove q from Q;
18: End while
19: Return Ordered_list
```

3.2 Initial Resources Allocation

When our proposed method optimizes individual queries, according to one of the orderings of Section 3.1, each query is given an initial resource allocation of CPU and memory that depends on the total resources available and the estimated resource allocation available for each query. This is to ensure that when there is a shortage of CPU (memory) resource, optimization of the individual query will take this into an account and use BJtrees (MJoins) as opposed to MJoins (BJtrees).

Algorithm 4 allocates CPU and memory resources for each query. Begin with a set of queries $Q = \{q_1, q_2, q_3, \ldots q_n\}$, where each is represented by an acyclic join graph. Query $q_i = (V_i, E_i)$ with vertices V_i from the set of streams $V_i \sqsubseteq V$. Given edge e = (K, L) as a predicate of q_i, with $K, L \in V_i$, let the join selectivity of e_j be $\sigma_{K,L}$. Let λ_K and λ_L denote the arrival rates of streams K and L, respectively, in a unit time. Moreover, number of edges / predicates in q_i is $|E_i|$. Let W_i be the window size of input streams of the query q_i. As all arriving tuples must be processed, we base the initial allocation of CPU and memory resources on the arrival rates and join selectivity as follows:

$$allocatedCPU\ (q_i) \quad \frac{\sum_{(K,L) \in E_i} \lambda_K * \lambda_L * \sigma_{K,L}}{\sum_{q_i \in Q} \sum_{(K,L) \in E_i} \lambda_K * \lambda_L * \sigma_{K,L}} * CPUavail \qquad (3\quad 4)$$

$$allocatedMEMORY\ (q_i) \quad \frac{W_i + \sum_{(K,L) \in E_i} \lambda_K * \lambda_L * \sigma_{K,L}}{(\sum_{j=1}^{n} W_j) + \sum_{q_i \in Q} \sum_{(K,L) \in E_i} \lambda_K * \lambda_L * \sigma_{K,L}} * MEMORYavail \qquad (3\quad 5)$$

The allocation of resources is based, in the numerator, on the arrival rates of streams involved in joins and on their selectivities. The denominator ensures that the total allocation of the resource over all queries does not exceed the constraints on the resource. Thus, the sum of resources allocated to each query equals to the total available resources.

If a join operation consumes less CPU than its allocated CPU resources, the unused CPU resources are provided to the other operations. Similarly, memory size that is unused is provided to the other join operations that require more memory.

3.3 Subexpression Sharing

The proposed algorithm generates qualified subplans that can be shared among different queries. Given a set of queries, there is a potential for the same join predicates to exist across queries. It is beneficial to utilize the similarity between queries by generating operations that are shared by queries. Therefore, this thesis introduces a sharing execution strategy that helps in keeping the CPU and memory costs below the available system resources.

A basic strategy is to evaluate each query individually on the incoming streams. However, if the similarity between queries is not exploited, this strategy can perform some redundant work as it may re-evaluate the same predicate more than once, which increases the overall evaluation cost. Moreover, collectively optimizing queries without sharing common join operations can negatively impact the performance of multiple query optimization. Therefore, choosing a sharing execution strategy for a set of queries is obviously an advantageous alternative compared to not sharing results.

To share the results of common subexpressions, the following factors should be taken into account:

The first factor is the sharing degree, which is adopted from Park and Lee (2012). Each join operation has a sharing degree, which indicates the reusability of the operation in the plan. The sharing degree of a join operation J_x in plan P is calculated as follows:

$$\xi(J_x|P) = \frac{1}{W_{max}} \sum_{q_j \in S(e)} W_j \text{, where } W_{max} = \max(W_j),\ w_j \text{ window for } q_j \text{ where } q_j \in e \qquad (3-6)$$

S(e) is the set of queries q_j that share the results of the join represented by edge $e \in E_i$. Recall that W_j denotes the window size of input streams of the query q_j. A join operation J_x is called a sharing operation if ξ $(J_x|P) > 1$. The greater the value of a sharing degree, the more a join operation can be shared and thus eliminate re-computing the same operation repeatedly and reducing the consumption of system resources. The main objective here is to find join operations that can be shared by incoming queries.

The second factor is the effective cost. Instead of considering the actual cost of a join operation, which is determined by equations (2-1) to (2-4), the effective cost is used when generating a qualified global plan. The following equations show how to calculate the effective cost of a sharing join operation J_x in a qualified plan P that is under consideration.

$$effCostRESOURCE(J_x \mid P) \quad \frac{actualCost(Jx|P)}{f(J_x, P)} \qquad (3 \quad 7)$$

In the above equation, in "effCostRESOURCE", where "RESOURCE" stands for "CPU", if the resource is CPU, while it is "MEM" if the resource under consideration is memory. The actualCost (j_x | P) is obtained by equations (2-1) and (2-2) in Section 2.1.5, if the operation is a BJtree, while it is obtained by Equations (2-3) and (2-4) if the join operation is an MJoin. The effective CPU cost of a subplan *sp* is the summation of the effective CPU costs of all operations in *sp*. Similarly, the effective memory cost of a subplan *sp* is the summation of the effective memory costs of all operations in *sp*.

$$effCostCPU(sp) \quad \sum_{Jx \in N_{sp}} effcostCPU\ (Jx \mid sp) \qquad (3 \quad 8)$$

$$effCostMem(sp) \quad \sum_{Jx \in N_{sp}} effcostMem\ (Jx \mid sp) \qquad (3 \quad 9)$$

In the above equations, N_{sp} is a set of nodes, i.e., joins in the subplan *sp*. As stated before, we will be generating a plan for a query at a time and incorporating it into a global plan. When searching for a plan for one query, different subplans will be examined and compared to each other to select the best one. To compare two subplans being considered in optimizing a query, we shall use equation (3-10) and Table (3-1). Let

M … be the total memory resource available,

M_g … be the memory consumed by the global plan g,

C … be the total CPU resource available, and

C_g … be the CPU resource consumed by the global plan g.

Then, the effective CPU and memory costs of *sp* relative to global plan *g* are combined as fractions into a tuple, denoted as *effCost*:

$$effCost(sp,g) = \left((effCostCPU(sp) + C_g) / C,\ \left(effCostMem(sp) + M_g\right) / M \right) \quad (3-10)$$

Equation (3-10) defines the effective cost of a subplan *sp*, *effCost*(*sp, g*), as a tuple (*c, m*) with $0 \le c, m \le 1$, where *c* is the effective CPU cost of *sp* plus the CPU cost of the global plan *g*, divided by the total CPU resource available. *m* is defined similarly using memory rather than CPU cost. More abstractly, *c* is the fraction of total CPU consumed by *sp* and global plan *g*, while *m* is the fraction of memory consumed by *sp* and global plan *g*.

When optimizing a query, an iterative process is used when choosing the next operation to be incorporated into the query's plan. Iterations are used to examine alternative subplans and then choose the best one. In each iteration, the current minimum plan is compared to a new plan under consideration, and if the new plan is better than the current minimum plan, the new plan becomes the current minimum plan. To determine which of two plans under consideration is better, we need to develop a relationship, which defines when one plan is better than another.

Let *sp1* and *sp2* be two subplans and let *effCost(sp1, g) = (c1, m1)* and *effCost (sp2, g) = (c2, m2)* relative to a global plan *g*. The comparison (*effCost*(*sp1, g*) < *effCost*(*sp2, g*)) in Lines(14, 39) of algorithm 4 is defined in Table 3-1 as:

Table 3-1 Definition of Less Than Comparator for Two (Sub)Plans

Values of m1, m2, c1, c2	**Result of (*effCost*(*sp1,g*) < *effCost*(*sp,g2*)**
m1 < m2 and c1 < c2	TRUE…………………… (1)
m1< m2, c1 > c2, $((c1+C_g)/C < (m1+M_g)/M)$	TRUE…………………… (2)
m1 > m2, c1 < c2, $((m1+M_g)/M < (c1+C_g)/C)$	TRUE…………………… (3)
none of the above hold	FALSE…………………… (4)

Recall that *C* and *M* represent the total CPU and the total memory resources available, respectively; C_g and M_g are CPU and memory consumed by the global plan formed thus far, respectively. Let *c2* and *m2* represent the respective effective CPU and memory costs of the current min subplan and we are considering a new subplan (with *m1* and *c1*) and are trying to decide whether it is better than the current min plan. Consider the conditions in Table 3-1. The new subplan is better if (1) holds as both the CPU and memory costs of the plan *sp1* are lower than those of *sp2*. The new subplan is also better if (2) holds: if the new subplan uses less memory than the current minimum subplan but uses more CPU, and memory is scarcer than CPU, then the new subplan is better - as defined by (2). The reasoning for the CPU is similar and is defined by (3).

The above factors are important for generating operations that can be shared by incoming queries. If there are two or more plans that are qualified for a query, using the above factors helps in deciding which query plan should be added to the global plan. An important point to be considered is to share operations that already exist in the global plan. Thus, the utilized algorithms, in our method, that generate MJoin and BJtree operators were modified to consider the sharing. The last two subsections in this chapter explain the utilized algorithms.

3.4 Global Plan Generation Algorithm

Algorithm 4 shows the pseudo-code of generating a global qualified plan. One of the inputs of the algorithm is queries. Each query is defined on a set of input streams and is represented by a join graph, which contain edges *E* and vertices *V*. A BJtree operator is generated by removing an edge *e* and combining two vertices. Figure 3-2 depicts a query and a generated BJtree operator.

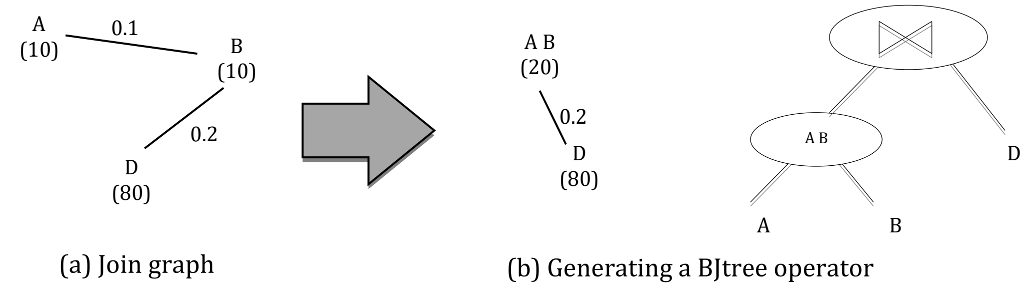


Figure 3-2 A query and a BJtree operator

There are two other inputs to Algorithm 4: the total available CPU and the total available memory. In the following section, our proposed technique is explained in more detail.

Recall that a plan *P* is represented by a set of nodes that are connected by a set of edges, and a subplan *SP* is a subgraph of a plan such that all nodes in *SP* are connected. We extend the Two-Layer JTree-Finder approach proposed in (Zhu et al., 2010) to generate a global plan for multiple queries. The Two-Layer JTree-Finder approach of (Zhu et al., 2010) is designed to generate qualified plans for a single query; thus, it does not consider producing an optimized plan for multiple queries. Our proposed method generates a global plan for multiple queries while exploiting the sharing of results of operations across queries. Algorithm 4 uses methods/subroutines, listed in Table 3-2, that perform specific tasks or implement algorithms, where algorithms 5, 6, and 7 are described later in this chapter. Also, some of the algorithms are based on works in other references as noted in the table.

Table 3-2 Subroutines Accessed by Algorithm 4

Subroutines	Description
Finding_Order_of_Queries(Q)	Implements Algorithm 3: Given a set of queries, return a sequence of the queries that satisfies an ordering criterion as defined by Algorithm3.
Find_MJoin_Ordering(*G*)	From (Viglas et al., 2003). Find MJoin_Ordering takes join graph of query *qi* as input and returns a subplan *sp* that joins all input streams in a single step with a specific order in an MJoin and thus also determining the specific sequence in which streams are joined within the MJoins.
State_Selection (sp, G, P)	Implements Algorithm 5. Inputs of State_Selection algorithm are a subplan sp, join graph g and the global plan P. The output consists of a subplan with both MJoin and binary join operators and an edge. In each iteration of State_Selection algorithm, an edge *e* from *G* is removed to generate a binary join operator.
Find_BJTree_Ordering (*G*, *P*)	Implements Algorithm 6. Inputs are a join graph *G* and the global plan *P* created thus far and the output is a subplan with binary join operators.
State_Removal (*sp*, c, *G*, *P, S*, min_ratio)	Implements Algorithm 7. Inputs of State_Removal algorithm are subplan *sp* and its actual CPU cost *c*, join graph *G*, the global plan *P*, a state *S* and min_ratio. The output is a subplan with both MJoin and binary join operators.
CPU(*sp*)	The actual CPU resources for subplan sp
Memory(*sp*)	The actual memory resources for the generated subplan sp by the adapted algorithms

Algorithm 4 The Global Plan Generation Algorithm

Input: (1) queries $Q = \{ q_1, q_2, \ldots, q_n \}$, where each query q_i is a Join Graph, over streams, with information on input rates, selectivities, and window sizes of query streams;
(2) CPU_{avail},
(3) $Memory_{avail}$

Output: A complete qualified plan P or -1 if a global qualified plan was not found

```
1: Initialize Global Plan P to an empty plan;
2: Q_ordering = Finding_Order_of_Queries(Q)
3: **For** i = 1 to n **do**
4: // First Layer
5:     Let $G_i$ be the join graph of query $q_i$, which is the i-th query in the ordering Q_ordering
```

```
6:  sp = Find_MJoin_Ordering($G_i$)
7:  If CPU(sp) $\leq$ allocatedCPU($q_i$) then
8:      If Memory(sp) $\leq$ $Memory_{avail}$ then
9:          $sp_{min} = sp$
10:         E = Set of edges in $G_i$
11:         While (E != null)
12:             (sp, e) = State_Selection (sp, $G_i$, P)
13:             If (Memory(sp) $\leq$ allocatedMEMORY($q_i$)) AND (CPU(sp) $\leq$ allocatedCPU($q_i$)) then
14:                 If effCost(sp, P) < effCost($sp_{min}$, P) then
15:                         $sp_{min} = sp$
16:                 End if
17:             End if
18:             Remove e from E
19:         End while
20:     Else // Memory(sp) > $Memory_{avail}$ – the subplan here is an MJoin subplan
21:         return -1 // no MJoin subplans can be found. Thus no qualified subplans can be found
22:     End if
23: Else // CPU(sp) > allocatedCPU($q_i$)
24: // Second Layer
25:     sp = Find_BJTree_Ordering($G_i$, P)
26:     If Memory(sp) $\leq$ allocatedMEMORY($q_i$)  then
27:         If CPU (sp) $\leq$ $CPU_{avail}$ then
28:             $sp_{min} = sp$
29:         Else // CPU (sp) > $CPU_{avail}$ - the subplan here is a BJtree subplan
30:             return -1 // no BJtree subplans can be found. Thus no qualified subplans can be found
31:         End if
32:     Else // Memory(sp) > allocatedMEMORY($q_i$)
33:         S = set of all intermediate states in subplan sp
34:         min_ratio = $\infty$
35:         For each state $s \in S$ do
36:             (sp, min_ratio) = State_Removal (sp, CPU(sp), $G_i$, P, s, min_ratio)
37:             If (Memory(sp) $\leq$ $Memory_{avail}$) AND (CPU(sp) $\leq$ $CPU_{avail}$) then
38:                 If effCost(sp, P) < effCost($sp_{min}$, P) then
39:                     $sp_{min} = sp$
40:                 End if
41:             End if
42:         End for
```

```
43:         End if
44:     End if
45:     Insert $sp_{min}$ into $P$
46:     Memory$_{avail}$ = Memory$_{avail}$ - Memory($sp_{min}$)
47:     CPU$_{avail}$ = CPU$_{avail}$ - CPU($sp_{min}$)
48: End for
49: return $P$
```

The algorithm first determines the order of queries to be optimized and included in the global plan. Line 2 invokes algorithm 3 described in Section 3.1.3 to determine the order of queries. Lines 3-48 form a loop that iterates over queries in the order as defined by the Q_ordering, which was determined in line 2. Thus, i in the loop, identifies the *i*-th query in the order Q_ordering.

At the beginning, the query under consideration (query q_i) is allocated a portion of the total available resources. This is so that the subplan generation within the loop does not produce mostly MJoins, and thus exhaust CPU resources, or binary joins, and thus exhaust memory. For query q_i, algorithm 4 produces an optimal MJoin subplan using the MJoin-ordering algorithm (Viglas et al., 2003) in the first phase (Line: 6). If the generated MJoin subplan does not use more CPU than is allocated to it and it does not use more memory than is available, then it is placed as the min subplan sp_{min} (Line: 9). The allocated CPU cost is used to ensure that CPU resources are not exhausted (Line: 7). Similarly, it is ensured that the subplan sp does not use more memory than is available (Line: 8). However, since sp was found using MJoin, which is more CPU intensive and does not use much memory, the State Selection algorithm is used to find a better subplan (Lines: 11-19). A qualified subplan with a better effective cost, which is determined using equation (3-10), is incorporated in the global plan (Line: 45). In case there is sufficient memory, the required CPU resources are checked and if the required CPU resources are more than allocated, finding a qualified subplan is still feasible and hence the second layer is triggered (Line: 24).

The second layer initially generates an optimal BJtree subplan (Line: 25). If the generated BJtree subplan requires more CPU than is available (Line: 29), qualified subplans can no longer be found because BJtree subplans have the least amount of CPU usage. BJtree subplans do not require the re-computing of intermediate results since they are stored in memory (Kossmann and Stocker, 2000). If the generated BJtree subplan requires more memory resources than the allocated resources, the State Removal algorithm is used (Line: 36). The State Removal algorithm may produce a qualified subplan with a better effective cost, which is determined using equation (3-10), (Lines 35-42). min_ratio, in lines 34 and 36, is a factor that helps in deciding whether to remove a state. Recall that a state means an intermediate result stored in memory. The min_ratio, in Algorithm 7, is calculated by subtracting the CPU cost of a subplan before removing a state from the CPU cost of the subplan after removing the state, and then it divides the result with the size of the memory saved after the state removal. Ultimately, a complete final qualified plan is returned.

One of the most significant benefits of using the Two-Layer Framework is that early termination can be achieved when either a qualified subplan is discovered or when the threshold limit for the required system resources is exceeded by all generated subplans in the first part of both layers. For example, when the required memory of the optimal MJoin subplan is larger than the available memory resources, then a qualified subplan can no longer be found as MJoin subplans have the least amount of memory usage. On the other hand, early termination is achieved if the required CPU resources of the generated BJtree subplan are more than the available resources.

3.5 Qualified Subplan Generation

Searching the extended plan search space exhaustively to generate a qualified execution plan is exponential in time complexity, which makes it not applicable to a data stream environment even with pruning techniques (Zhu et al., 2010). Therefore, our Algorithm 4 extends the Two-Layer JTree-Finder algorithm introduced in (Zhu et al., 2010) that generates a qualified plan for a single query in polynomial time. We adapt it so it works on multiple queries. Our proposed algorithm utilizes four different algorithms: the MJoin-ordering algorithm (Viglas et al., 2003), BJtree algorithm (Kossmann and Stocker, 2000), State Selection algorithm and State Removal algorithm (Zhu et al., 2010). However, three of them are modified, in our work, to exploit the sharing of join operators in a multi-query environment.

3.5.1 First Layer Algorithms (MJoin and State Selection)

3.5.1.1 MJoin Ordering Algorithm

The first layer of the proposed approach triggers the MJoin-ordering algorithm that is adopted from (Viglas et al., 2003). This algorithm generates a subplan that joins multiple streams in a single step without storing intermediate results. When a new tuple arrives, it joins with the other streams in a specific order. The join order is determined by considering the input rates of each stream and join selectivity of a join operation. The MJoin algorithm selects the next input to join if it generates the smallest number of intermediate results. The number of intermediate results (i.e. weight) is the product of the join selectivity and the arrival rates of the operand streams: $\lambda_a * \lambda_b * \sigma_{a,b}$. In each iteration, this algorithm selects one stream and sets it as a root. Then, the weight of other streams is calculated to find the join sequence of the rooted stream. For each rooted stream, the time complexity to calculate the weight of streams is O(n), where *n* is the number of streams. For all rooted streams, the time complexity to find the weight of streams is O(n^2). To illustrate, consider a query over *n* input streams. Each input stream needs to join with *n*-1 streams. In Figure 3-3(a), for example, if a tuple arrives at stream *B*, it joins with tuples from stream *A* since joining *A* and *B* produces the smallest number of intermediate results. The generated results then join tuples from stream *D* because it produces the second smallest number of intermediate results. The time complexity for joining streams is O(log (n)). The total time complexity to find the join ordering for each stream is O(n log (n)), whereas finding the join sequence for all input streams has a time complexity of O(n^2 log (n)). The best MJoin ordering is the least CPU costs.

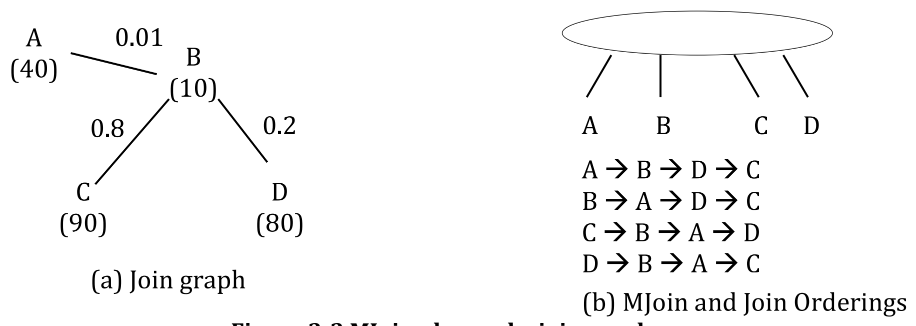


Figure 3-3 MJoin plan and a join graph

We use this algorithm without any modification because it is not affected by the existence of operators in the global plan. This is because it generates one MJoin operator where all input streams of a query are joined in a single step. For example, consider two queries: the first query needs to join streams *A*, *B* and *C*, whereas the second query needs to join streams *A*, *B* and *D*. Since the intermediate results are not stored, both queries will not share any subplan even though both queries are partially similar. The goal here is to reduce the CPU consumption without increasing the usage of memory resources. Figure 3-3(b) shows an optimal MJoin plan for the join graph presented in Figure 3-3(a).

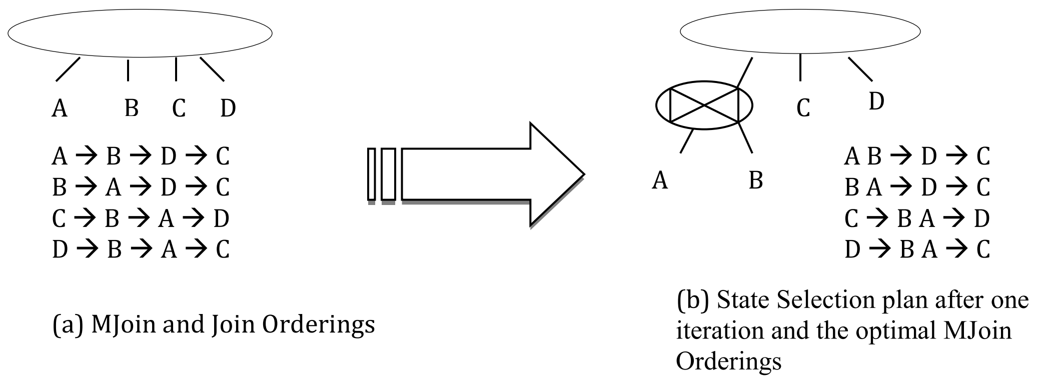


Figure 3-4 MJoin ordering and a State Selection plan

3.5.1.2 State Selection Algorithm

The second algorithm, Algorithm 5 is State Selection and it is adapted from (Zhu et al., 2010). The original algorithm was proposed to optimize a single query and thus we needed to modify it so that the sharing of results across multiple queries would be exploited. The inputs to this algorithm are a join graph *G,* which is a graph of one of the queries originally input, the global plan *P* created thus far, and an MJoin subplan generated by the MJoin ordering algorithm. The three fundamental factors that determine which intermediate states to store are the existence of the state in the global plan, the edge weight and the edge frequency. An intermediate state is the result of a join operation that is stored in the memory. Each edge in *G* creates a binary join when it is removed. If a result of one of the join operations already exists in the global plan, its cost is considered to be zero for the query under consideration - thus that join will be used/adopted (Lines: 3-7 of algorithm 5). In case there are no join operations in the global plan that can be used by the query, we use the same process that is used in (Zhu et al., 2010). The edge with the largest ratio of frequency/weight is used to identify the query's binary operator (Line: 9). Recall that the weight of each join predicate is determined by multiplying the join selectivity with the arrival rates of the operand streams (λ_A* λ_B * σ_{AB}). The frequency is determined by how many times two streams are joined after each other in the join orderings as shown in Figure 3-4(a). Thus, a binary operator is selected either because it already appears in the global plan or it has the highest frequency/weight ratio. After creating a binary join operator whose output is now fed to the MJoin operator such as in Figure 3-4 (b) (but now for an MJoin with one less input stream) we use the MJoin-ordering algorithm from (Viglas et al., 2003) with a time complexity of O(n2 log (n)) (Lines: 5, 11 of algorithm 5). The State Selection algorithm has a time complexity of O(n + n^2 log (n)) (Zhu et al., 2010). Figure 3-4 (b) shows a State Selection plan. Note that a global plan *P* contains a set of subplans and any subplan that is an element of *P* can answer a part or a whole query.

Algorithm 5 State Selection Algorithm

```
Input:  1) Join Graph G=(V, E) of a query.
        2) Global Plan P,
        3) Subplan sp, such that its final operation is an MJoin that produces the subplan's result
Output: 1) Subplan $SP_{new}$
        2) Selected edge e, such that $SP_{new}$ generates the same result as the given plan sp and operations of
        $SP_{new}$ are the same as that of sp except that a binary join is created over the output edge e
1: Let $P_{oper}$ be the set of operations (joins) in P
2: For e = ($v_x$ , $v_y$ ) $\in$ E do
3:     If (exists operator o $\in P_{oper}$ that joins two streams in G connected by e) then
4:         $G_{new}$ = Graph G that is modified by removing edge e and coalescing vertices $v_x$ and $v_y$ into $v_{xy}$;
5:         $SP_{new}$ = Find MJoin Ordering ($G_{new}$);
6:         Return ($SP_{new}$, e)
7:     End If
```

```
8: **End For**
9: Choose edge e = ($v_x$ , $v_y$ ) $\in E$ that has the max ratio of (frequency/weight)
10: $G_{new}$ = Graph $G$ that is modified by removing edge $e$ and coalescing vertices $v_x$ and $v_y$ into $v_{xy}$;
11: $SP_{new}$ = Find MJoin Ordering ($G_{new}$);
12: **Return** ($SP_{new}$, $e$)
```

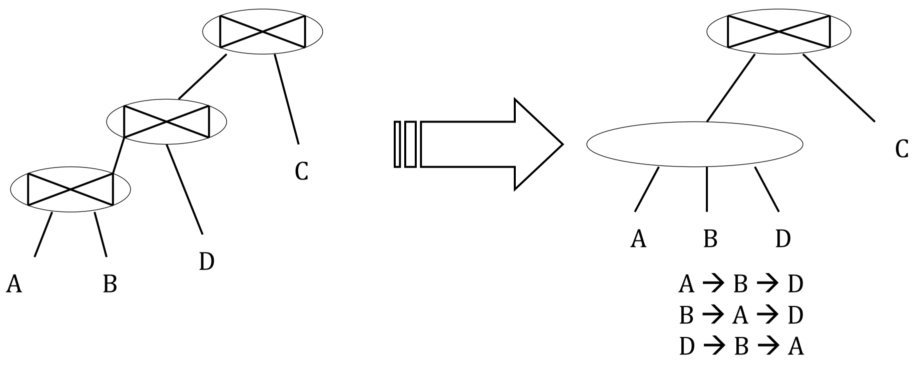
3.5.2 Second Layer Algorithms (BJtree and State Removal)

3.5.2.1 BJtree Algorithm

The BJtree algorithm, Algorithm 6 is triggered in our second layer. Unlike the MJoin-ordering algorithm, this algorithm takes multiple steps to join input streams. Intermediate results are stored when using the BJtree algorithm. To elaborate, when new tuples arrive from an input stream, they join with the tuples from another input stream. The joined tuples are stored in an intermediate state, and tuples in this intermediate state then join with either tuples from a third input stream or tuples from other intermediate results. There are two factors that determine which intermediate states to store. The first is the existence of the state in the global plan, and the other factor is the edge weight. If an intermediate state already exists in the global plan, that is, if the binary join under consideration is already a part of the global plan, there is no reason to recalculate the result. The query that is being optimized will use the results of this intermediate state, at zero cost, as it already has been created for other queries (Lines: 4-8 of algorithm 6). After already existing results in the global plan are incorporated into the currently optimized query, the edge with the smallest weight is used to generate intermediate results in each iteration (Lines: 10- 15). This algorithm has a time complexity of O(n^2 log (n)).

Algorithm 6 BJtree Algorithm

```
Input: 1) Join Graph G=($V$, $E$) of a query.
       2) Global Plan $P$
Output: $SP_{new}$
1: $SP_{new}$ = empty,
2: Let $P_{oper}$ be the set of operations (joins) in $P$
3: For each edge $e = (v_x, v_y) \in E$ do
4:     If (exists operation $o \in P_{oper}$ that joins two streams connected by $e$ in $G$) then
5:         Add the operation $o$ to the subplan $SP_{new}$;
6:         Modify $G$ by coalescing vertices $v_x$ and $v_y$ into a vertex $v_{xy}$ and assigning $v_{xy} = 2 * \lambda_x * \lambda_y * \sigma_{x,y}$;
7:         Remove $e$ from $E$;
8:     End if
9: End For
10: while ($E$ != null) do
11:     Let operation $c$ be an operator that joins two streams connected by $e$ in $G$;
12:     Find that $e = (v_x, v_y) \in E$ that has the smallest weight (using eq. 3-1)
13:     Add the operation $c$ to the subplan $SP_{new}$;
14:     Modify $G$ by coalescing vertices $v_x$ and $v_y$ into a vertex $v_{xy}$ and assigning $v_{xy} = 2 * \lambda_x * \lambda_y * \sigma_{x,y}$;
15:     Remove $e$ from $E$;
16: End while
17: Return $SP_{new}$
```



(a) A BJtree plan

(b) State Removal plan after one iteration and the optimal MJoin Orderings

Figure 3-5 BJtree plan and State Removal

3.5.2.2 State Removal Algorithm

The State Removal technique presented in (Zhu et al., 2010) has been adapted for use in our second layer. The adaption deals with modifications to exploit the sharing of results across queries. As shown in algorithm 7, the inputs to this algorithm are a join graph, the global plan *P* created thus far*,* a subplan *sp*, CPU cost of *sp,* a selected state to be removed, and min_ratio which helps in deciding whether or not to remove the chosen state. To decrease the memory consumption, the State Removal algorithm eliminates some of the existing intermediate states. Thus, it provides a trade-off between CPU and memory resources.

There are three factors that need to be considered to remove intermediate results. The first is whether or not the intermediate state exists in the global plan. If it already exists, it will be used by the incoming queries as there is no additional cost in using it for this query. In case there are no intermediate states in the global plan that can be used by the query, we remove intermediate states by considering the other two factors as presented in (Zhu et al., 2010). The second factor is the amount of memory saved when a state is removed, whereas the third is the additional CPU costs required to re-compute the intermediate results (line: 11 of algorithm 7). Figure 3-5(b) shows a step of the State Removal technique, and Figure 3-5(a) represents the input BJtree plan. The State Removal algorithm has a time complexity of $O(n + n^2 \log (n))$ (Zhu et al., 2010).

Algorithm 7 State Removal Algorithm

```
Input: 1) Join Graph G=(V, E) of a query.
       2) Global Plan P.
       3) subplan sp, where sp is not part of P
```

```
        4) CPU($sp$);
        5) state $S_n$ in $sp$;
        6) min_ratio;
Output: TheReturnedSP, $min\_ratio_{new}$
1: $SP_{old} = sp$;
2: $min\_ratio_{old}$ = min_ratio.
3: Let $P_{oper}$ be the set of operations (joins) in $P$
4: Let $op1 \notin$ P be a join operator that produces $S_n$
5: Let $op2 \notin P_{oper}$ be any join operator that feeds tuples to $S_n$
6: Let $M$ be an MJoin with inputs as the union of inputs to $op1$ and to $op2$
7: $SP_{new} = sp$ modified by replacing the inputs of $op1$ and of $op2$ with $M$
8: Let $G' \in G$ be a subgraph of $G$ such that it contains those streams that are processed by $M$.
9: $SP_{new}$ = Find_MJoin_Ordering(G');
10: Let $z$ be the memory saved when removing $S_n$ from $sp$;
11: state_ratio = (CPU($SP_{new}$) – CPU($sp$)) / $z$
12: If (state_ratio < min_ratio)
13:     Return $SP_{new}$, state_ratio
14: Else
15:     Return $SP_{old}$, $min\_ratio_{old}$
16: End if
```

CHAPTER 4 : EXPERIMENTAL EVALUATION

In this section, the proposed method is evaluated to examine its effectiveness. All the algorithms in this thesis were implemented in Java, and all the experiments were executed on a Core 2 Duo CPU 2.4 GHz system with 10 GB RAM.

In the experiments, a set of multi-way join queries is randomly generated. For each query, n input streams are chosen from 20 sources of streaming data ($n \leq 20$). Since equi-join queries are popular in wireless sensor networks, all queries in the experiments are equi-join queries with (n-1) join predicates. Table 4.1 represents the parameters that govern the experiments, and their values.

The input rate of each input stream is randomly generated over the range [10…. 100] (tuples/s). The join selectivity of each join operation is randomly generated in the range of [0.01, 0.2]. The window size of each query is set to be the same at 10 seconds. Input rates and join selectivities have a uniform probability distribution over the specified range. Values used in this experiment are typical values and they are used by other researchers (Hammad et al., 2003) (Zhu et al., 2010) (Park and Lee, 2012).

Table 4-1 Experimental Parameters

Parameters	Values
Number of join predicates in each query	3, 5, 7, 9
Number of queries	10
Number of streams *n*	20

4.1 Experimental methodology

The objective of this experimentation is to evaluate the proposed method by comparing it to the performance of the Two-Layer JTree-Finder approach proposed in (Zhu et al., 2010). To optimize multiple queries, Zhu et al suggest that we first need to allocate resources to each query and then directly apply their Two-Layer JTree-Finder algorithm to each query. Each query is independently optimized by their algorithm without the sharing of results of common join operators - we refer to their optimization as the naïve approach since it was not designed for multiple queries. Moreover, we also compare our method with this naïve approach when the join operations are shared in the naïve approach - we refer to it as the Shared Naïve approach. Furthermore, we also evaluate our method that finds the order of queries in this chapter.

We compare the CPU and memory resource requirements for plans generated by our method, the naïve approach, and the naïve approach that shares operators under four different scenarios that vary in terms of the constraints on the CPU and memory resources. We also report on the execution delay to generate the plans. The second layer of the Two-Layer JTree-Finder approach will only be triggered if the first layer does not generate a qualified plan. To illustrate, Zhu et al., stated in their work that, " 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" (Zhu et al., 2010, p. 21). We use the Two-Layer JTree-Finder technique in the experiments in the same way that it is proposed.

The experimental sets are: 1) A data stream system has sufficient CPU and memory resources to execute the proposed method and the Two-Layer JTree-Finder algorithm, 2) there are constraints on both CPU and memory, 3) memory resources are restricted and CPU is configured to be sufficient for executing any approach, 4) the CPU resources are limited while having sufficient memory. When there are constraints on both CPU and memory, both the naïve approaches and our technique utilize the negative correlation between CPU and memory to find qualified plans.

Because the Two-Layer JTree-Finder approach only generates the local execution plan of each query individually, the time required to generate all the local plans of all the queries consecutively is measured.

There are a number of scenarios for which all queries cannot be processed due to exceeding resources. One scenario is to increase the number of queries with decreasing system resources. Moreover, increasing the arrival rates of input stream can result in exceeding system resources. Having high join selectivity, also, will increase the probability of exceeding resources.

To calculate the CPU costs, this thesis uses the same costs as the CAPE engine (Rundensteiner et al., 2004), which are measured also in (Zhu et al., 2010). Table 4.2 shows the average CPU costs of the basic tuple operations, namely joining (C_j), inserting (C_i), and deleting (C_d) operations in this engine, which are computed in milliseconds.

Table 4-2 CPU Costs of Basic Tuple Operations in (ms)

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

4.1.1 Sharing Ratio

When multiple queries are processed, there is a chance that there is some similarity between them. The level of sharing between queries is referred to as the sharing ratio and it can be calculated as follows (Park and Lee, 2012):

$$SR(Q) = \frac{1}{|J(Q)|} \sum \frac{\xi(J_i)}{|Q(J_i)|}$$

J(*Q*) denotes all distinct join predicates presented in queries *Q*. Moreover, $\xi(J_i)$ denotes the sharing degree of a join operation which is calculated using equation (3-6). $Q(J_i)$ denotes the set of queries that have the join predicate J_i. As the sharing ratio increases, the chance of there being shared join operators increases. In our experiments, we generate queries with 0.6 sharing ratio since this ratio is used in Park and Lee's research (2012).

4.2 Experimentation Results

4.2.1 Query Ordering

In the first experiment, we evaluate our method that generates the order of queries. Five join queries with 5 join predicates are used to find the best method that generates the order of queries. For each query, 6 input streams are chosen from 12 sources of streaming data. In this experiment, there are constraints on both CPU and memory. Figure 4-1 shows the CPU costs of the three proposed techniques: heaviest to lightest, lightest to heaviest, and the shared predicate technique, whereas Figure 4-2 shows the memory costs.

The shared predicate method consumes less CPU and memory resources than the other two methods as shown in Figures 4-1 and 4-2. This is because the shared predicate method takes into consideration the costs of join predicates inside queries, and it tries to find join operations that can be shared between all possible queries. In some cases, it may occur that the heaviest to lightest query technique requires the same or slightly more CPU/memory than the shared predicate technique. This happens when heaviest queries happen to have multiple join operations shared among different queries.

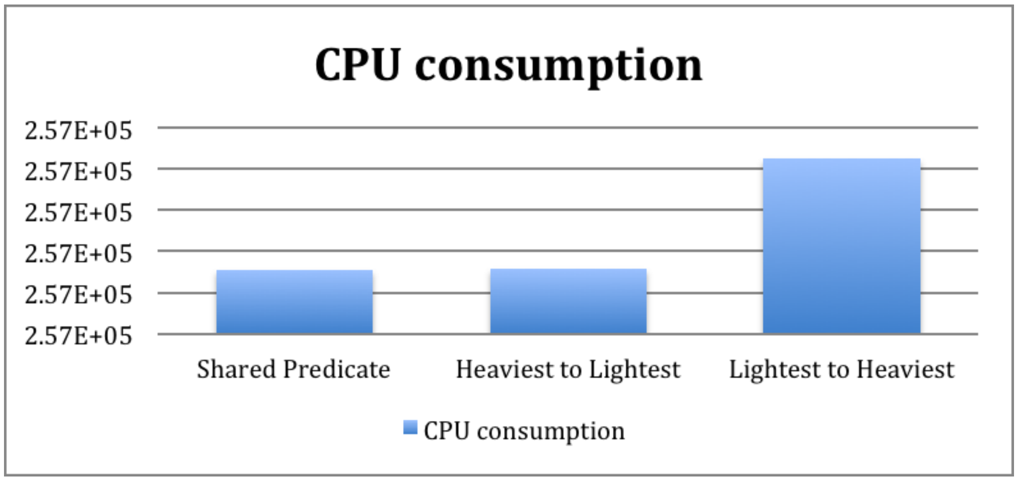


Figure 4-1 The CPU costs of the three proposed techniques

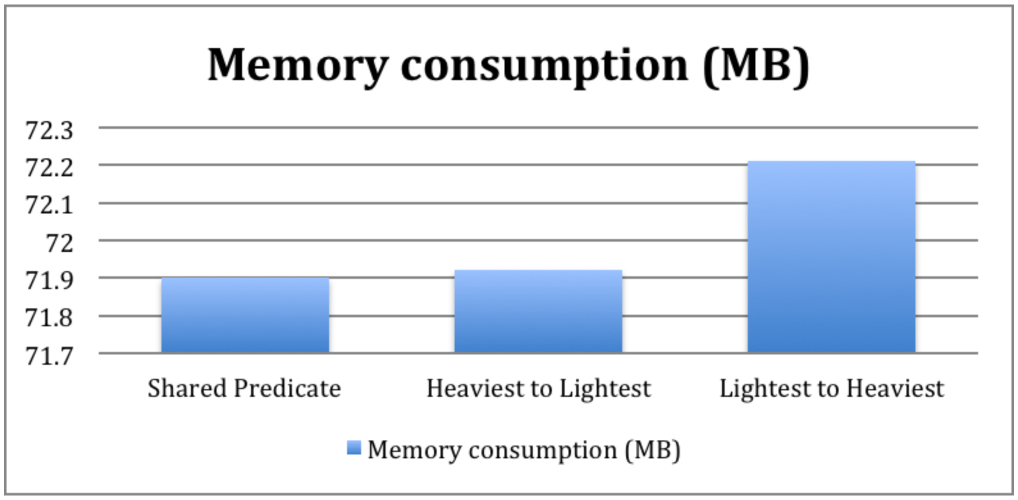


Figure 4-2 The memory costs of the three proposed techniques

Since the shared predicate technique is superior to the other two techniques, we use this technique in the rest of the experiments.

4.2.2 No Constraints on Resources

When a data stream system has more CPU and memory resources than it needs, the Naïve approach only generates MJoin plans for individual queries because the MJoin algorithm is the first encountered algorithm in its first layer. Our shared technique keeps searching for subplans that share join operations among different queries. Thus, subplans with smaller effective costs are generated, and they could be BJtree, MJoin or a mixture of both. Figure 4-3 shows the CPU costs of executing the plan of our technique, the Naïve approach, and the Shared naïve approach. Since all queries generated by the Two-Layer JTree-Finder approach have MJoin plans, both naïve approaches consume more CPU resources than our proposed approach. However, our proposed approach requires more memory than the both naïve approaches as shown in Figure 4-4. This is because our technique produces BJtree operators that can be shared among queries. In case we limit the memory resources, our method will mostly generate MJoin plans.

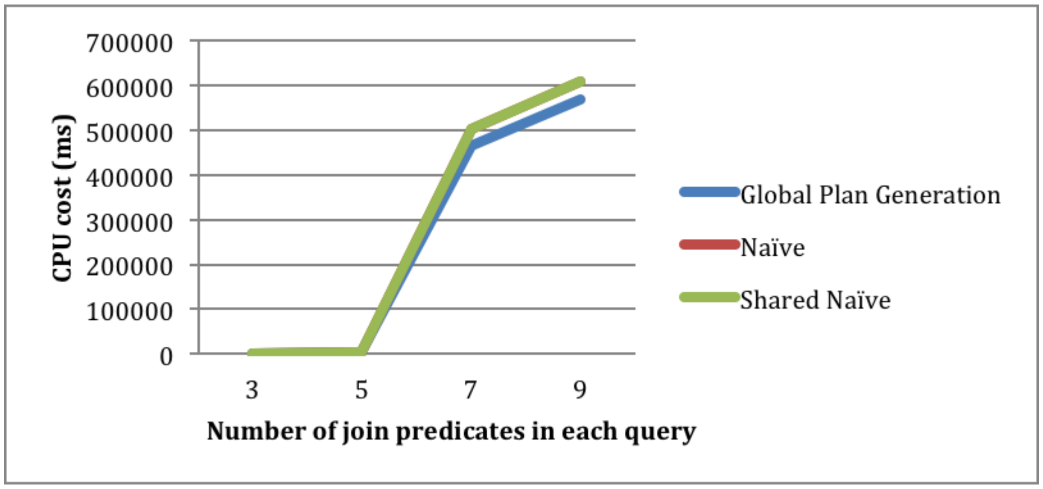


Figure 4-3 CPU consumptions of the three techniques – no constraints on CPU or memory

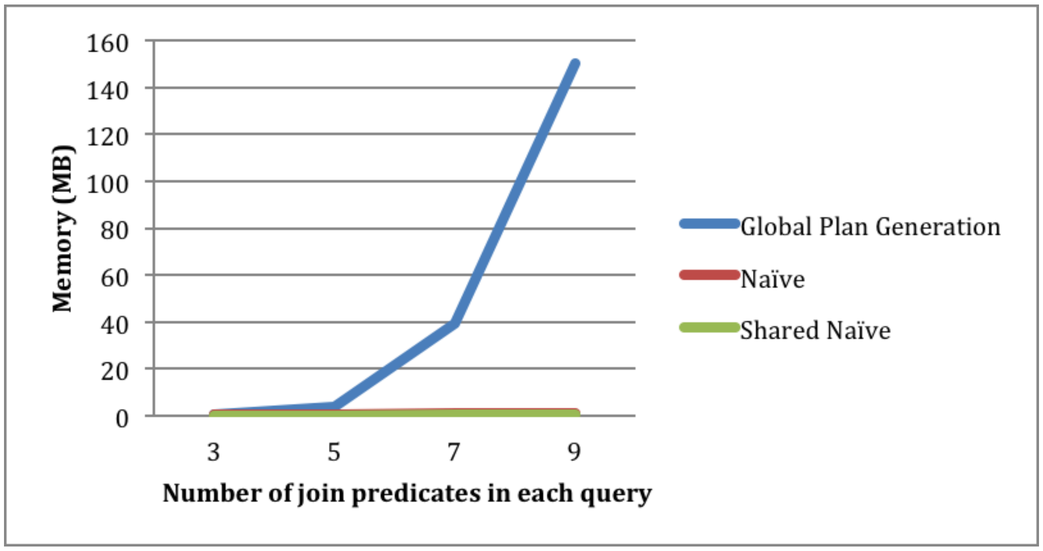


Figure 4-4 Memory consumptions of the three techniques – no constraints on CPU or memory

4.2.3 Both Resources Restricted

In the experiment set 2, the naïve approach uses either the State Selection algorithm or the State Removal algorithm to find a qualified plan. As shown in Figure 4-5, the CPU costs of our global plan are less than the CPU consumption of the plans generated by both naïve approaches. This is because our method searches for and exploits sharing in its optimization. Query plans sometimes include operators that are expensive for the query, but when amortized over a number of queries they are efficient. In contrast, the Naïve approach optimizes each query individually without sharing. The Shared naïve approach also first optimizes queries individually, but sharing is also facilitated by forming a global plan, from individual query plans, while not re-computing joins redundantly; thus it misses the situations where one combination of joins may be expensive for a single query but very beneficial across queries. Figure 4-6 shows the memory consumption of the three approaches. Plans generated by our method require less memory than the plans produced by both naïve approaches. This is because our method generates less intermediate states compared to the other two approaches.

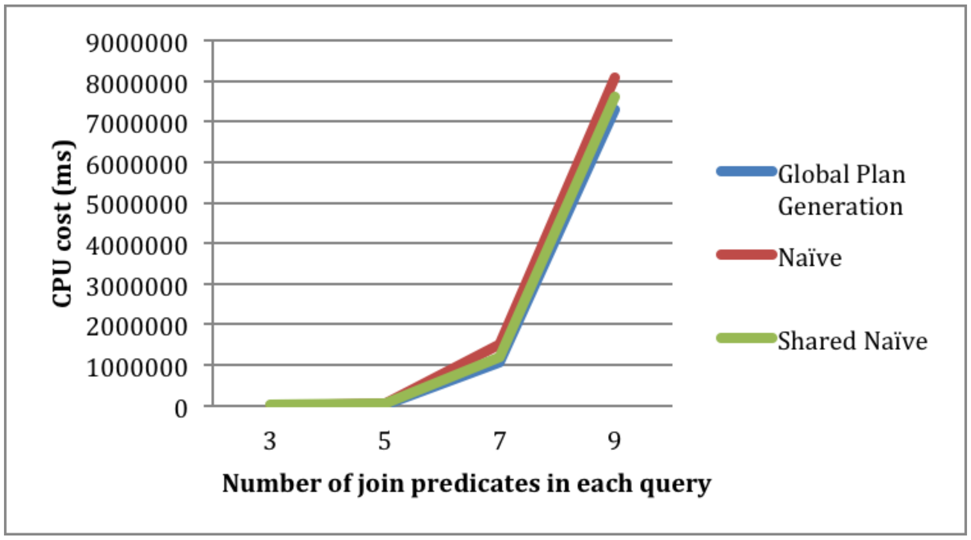


Figure 4-5 CPU consumptions when restricting both resources

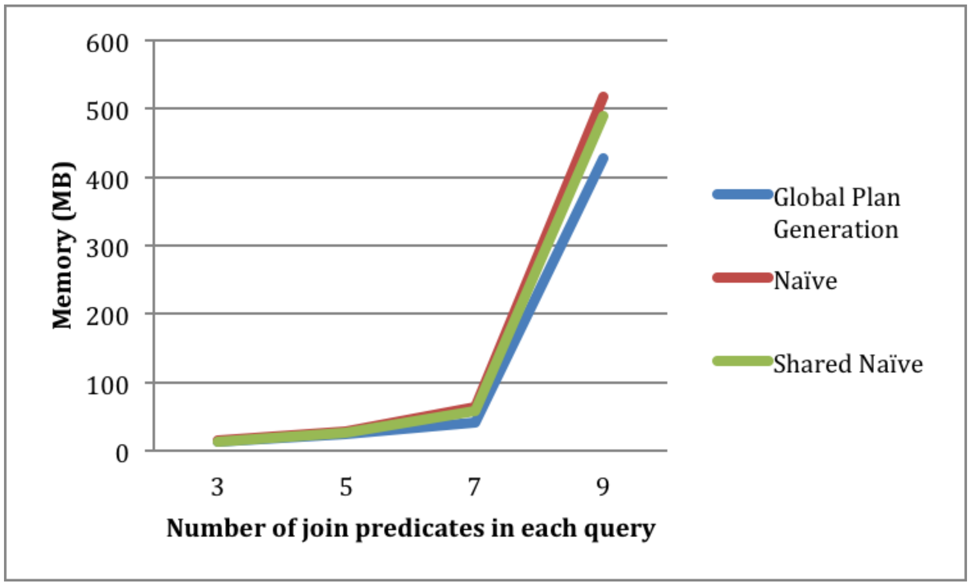


Figure 4-6 Memory consumptions when restricting both resources

4.2.4 Constraints on Memory Resources

In the experiment set 3, when memory resources are restricted and CPU is configured to be sufficient for executing any approach, all techniques act similarly to some extent. Most of the operations generated are MJoin because CPU resources are unlimited. Nevertheless, our shared technique sometimes generates a few BJtree operations, and that explains the reason for it consuming slightly more memory and less CPU resources compared to the naïve approach that shares operators as shown in Figure 4-7. All three techniques do not exceed the restricted resources that are assigned in the experiments. However, if we force our shared technique to generate plans that have the least possible amount of memory, then our technique will act the same as the naïve approach that shares operators.

When multiple queries overlap in the required data stream source, queries in the shared techniques access the same incoming stream state. In the Naïve approach, the queries do not access the same incoming stream states. In other words, each query has a state for each incoming stream source, even though some of the states may exist in the memory for another query. Therefore, the naïve approach consumes more memory compared to the two shared techniques. If the states of incoming streams are shared in the Two-Layer JTree-Finder technique, its memory usage will be almost the same level as for the Shared naïve approach. Figure 4-8 shows the memory usage.

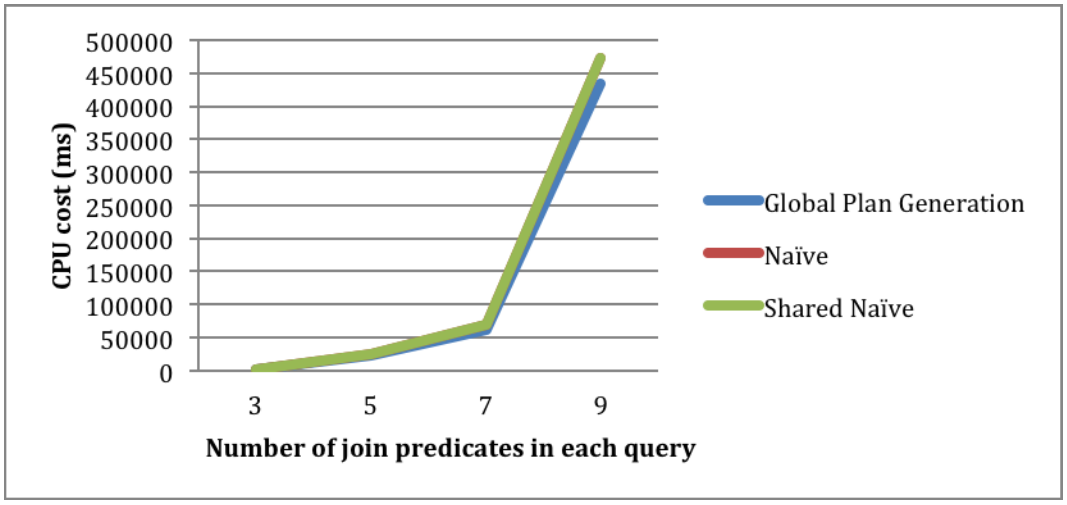


Figure 4-7 CPU consumptions when restricting the memory resources.

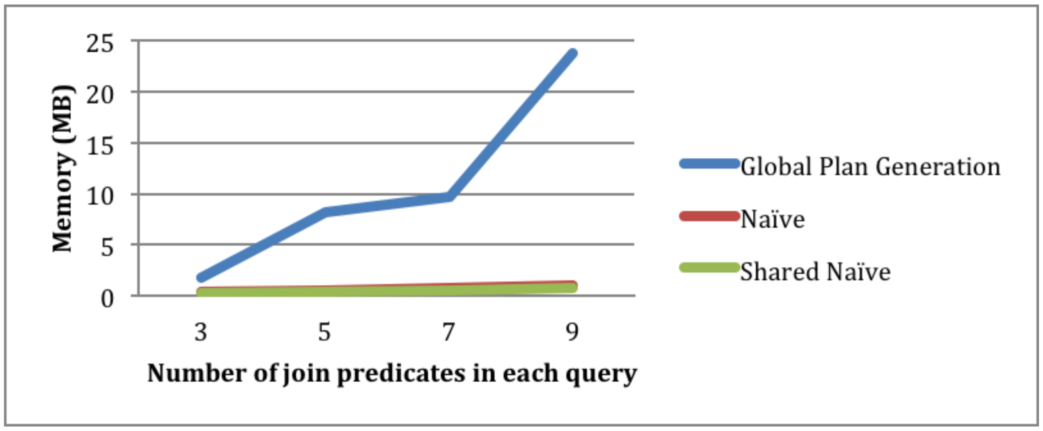


Figure 4-8 Memory consumptions when restricting the memory resources.

4.2.5 Constraints on CPU Resources

In the experiment set 4, when the CPU resources are limited while having sufficient memory, the naïve approaches always generate BJtree plans for queries. Similarly, our global plan generation technique will only produce BJtree operators. The CPU costs of the three techniques are shown in Figure 4-9. Our shared technique consumes less CPU resources than the naïve approach that shares operators. Moreover, as the number of join predicates increases, the two naïve approaches require more memory space than our proposed method as shown in Figure 4-10. The three techniques do not exceed the restricted resources that are assigned in the experiments.

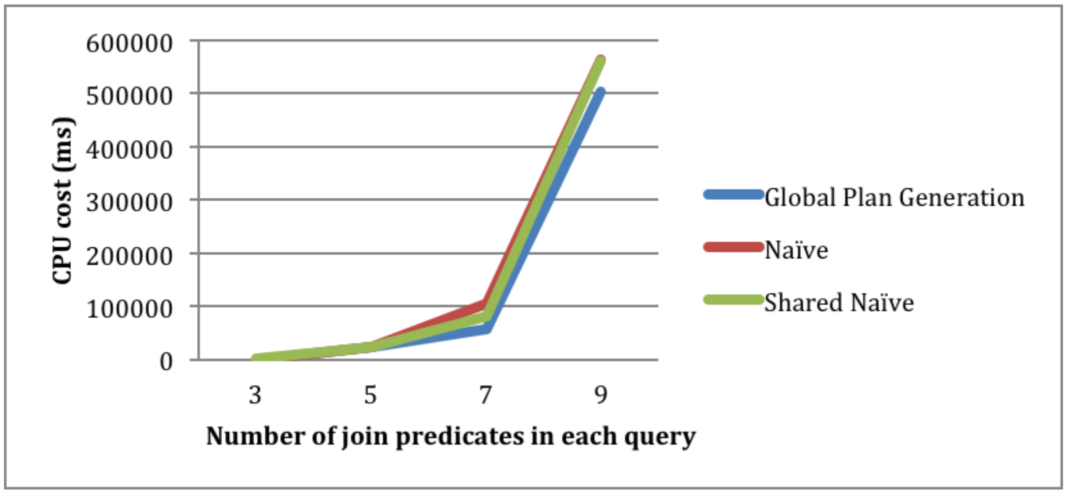


Figure 4-9 CPU consumptions when restricting the CPU resources.

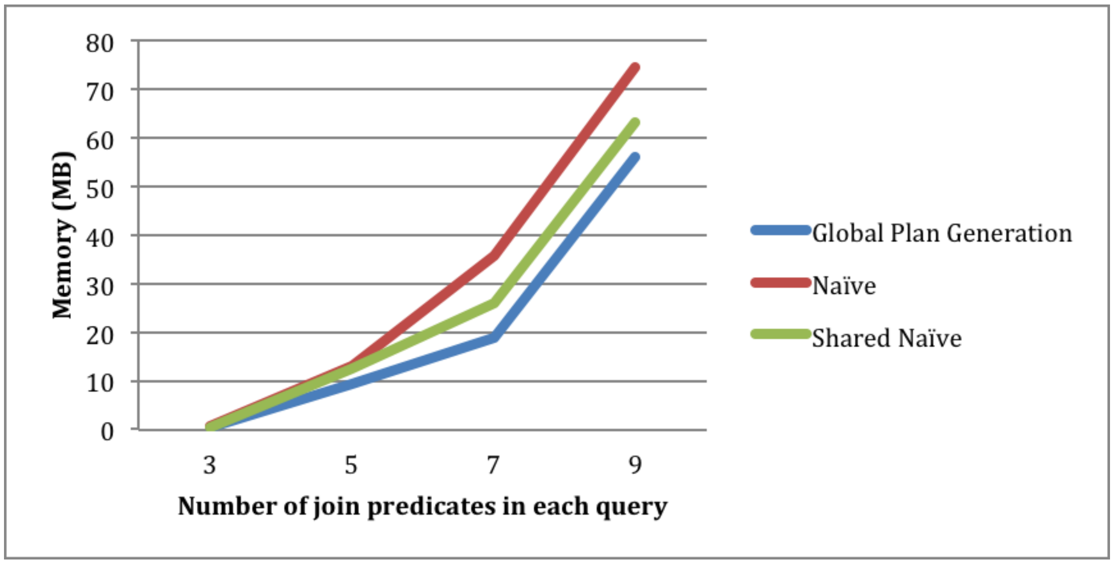


Figure 4-10 Memory consumptions when restricting the CPU resources.

4.2.6 Optimization Delays

In the next experiment, the execution delay (optimization delay) of the naïve approaches and our proposed method to generate the plans is measured. Ten join queries with 2, 4, 8, and 16 join predicates are used to generate plans using the two techniques. When having sufficient resources, the shared approach requires more time than the JTree-Finder because our shared technique keeps searching for join operations that can be shared among queries. On the other hand, the naïve approach returns the first qualified plan it encounters. Similarly, when one or both CPU and memory resources are limited, our shared approach requires more time to generate the global plan. Figure 4-11 shows the execution delay for both techniques. Both resources are limited when queries with 16 join predicates are evaluated. The CPU is restricted when queries with 8 join predicates are evaluated. However, queries with 4 join predicates are evaluated when the memory is limited. When we have sufficient CPU and memory, we evaluate queries with 2 join predicates.

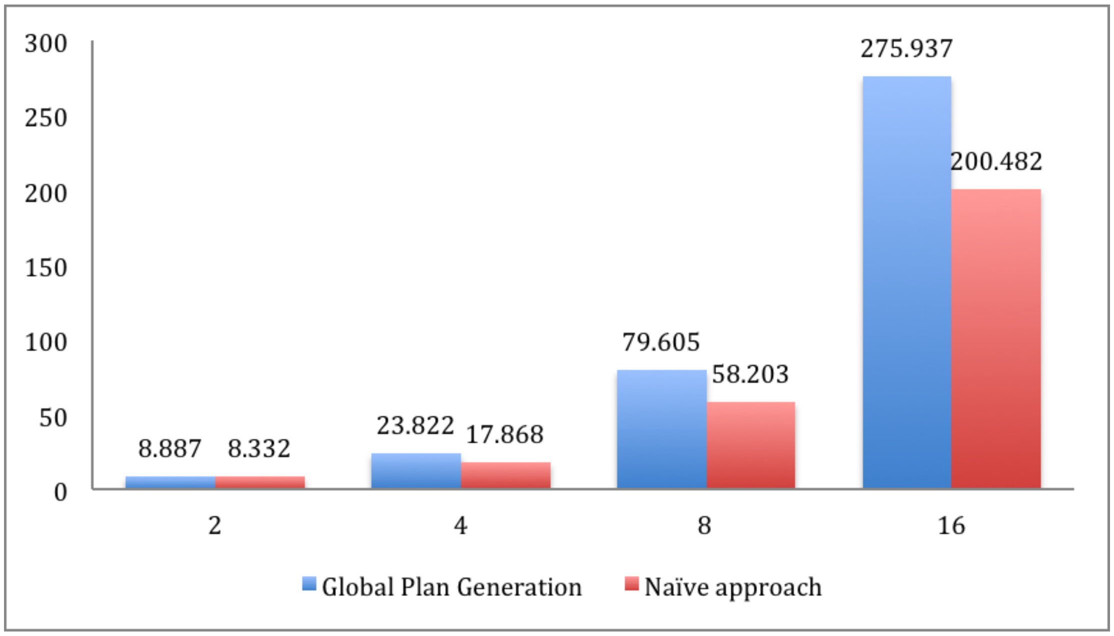


Figure 4-11 Execution time in ms required by the two techniques

CHAPTER 5 : CONCLUSION AND FUTURE WORK

In data stream applications, a set of multi-way join queries is registered to be processed in a continuous manner. A query execution plan needs to be adhered to both CPU and memory resource constraints because one of these resources being out of bound hinders processing incoming streams in a data stream system. The two basic methods used to execute queries over streams are BJtree and MJoin plans. The former consumes more memory resources than the latter, whereas the latter requires more CPU resources than the former. A third method to execute queries over streams is called JTree, which combines the BJtree and MJoin techniques. There is a trade-off between memory and CPU consumptions when using the JTree method.

This thesis introduces a technique that generates a qualified global plan for multi-join queries. It utilizes the BJtree, MJoin, and the JTree methods to produce plans. Our method creates the global plan by optimizing individual queries and adding their operators to the global plan: when optimizing a query, the proposed method checks the global plan for any join operations that can be reused or shared by the query. Moreover, our method allocates system resources to each query before processing it. This is to ensure that when there is a shortage of the CPU (memory) resource, optimization of the individual query will take it into account to use BJtrees (MJoins) as opposed to MJoins (BJtrees). Because our method optimizes each query individually, we proposed three new techniques to find the order of queries in which they need to be processed.

For future work, one avenue is to study and address challenges when trying to generate qualified plans for a distributed data stream system. Another avenue for future work is to consider monitoring statistics of all previously executed join operations to estimate their current execution costs and the correlation between CPU and memory. This can be very helpful because it may reduce the time of generating query plans.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J. H., ... & Zdonik, S. B. (2005, January). The Design of the Borealis Stream Processing Engine. In *CIDR* (Vol. 5, pp. 277-289).

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., ... & Widom, J. (2016). Stream: The stanford data stream management system. In *Data Stream Management* (pp. 317-336). Springer Berlin Heidelberg.

Avnur, R., & Hellerstein, J. M. (2000, May). Eddies: Continuously adaptive query processing. *In ACM SIGMoD Record* (Vol. 29, No. 2, pp. 261-272). ACM.

Babcock, B., Babu, S., Datar, M., Motwani, R., & Thomas, D. (2004). Operator scheduling in data stream systems. *The VLDB Journal—The International Journal on Very Large Data Bases, 13*(4), 333-353.

Babcock, B., Babu, S., Motwani, R., & Datar, M. (2003, June). Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 253-264). ACM.

Babcock, B., Datar, M., & Motwani, R. (2004, March). Load shedding for aggregation queries over data streams. In Data Engineering, 2004. *Proceedings. 20th International Conference on* (pp. 350-361). IEEE.

Babu, S., Motwani, R., Munagala, K., Nishizawa, I., & Widom, J. (2004, June). Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. 407-418). ACM.

Babu, S., Munagala, K., Widom, J., & Motwani, R. (2005, April). Adaptive caching for continuous queries. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on* (pp. 118-129). IEEE.

Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., & Stonebraker, M. (2003, September). Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29* (pp. 838-849). VLDB Endowment.

Chen, J., DeWitt, D. J., Tian, F., & Wang, Y. (2000, May). NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 379-390). ACM.

Chen, T., Chen, L., Ozsu, M. T., & Xiao, N. (2013). Optimizing multi-top-k queries over uncertain data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 25(8), 1814-1829.

Dalvi, N. N., Sanghai, S. K., Roy, P., & Sudarshan, S. (2003). Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, *66*(4), 728-762.

DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., & Wood, D. A. (1984). *Implementation techniques for main memory database systems* (Vol. 14, No. 2, pp. 1-8). ACM.

Golab, L., & Özsu, M. T. (2003). Issues in data stream management. *ACM Sigmod Record*, *32*(2), 5-14.

Hammad, M. A., Aref, W. G., & Elmagarmid, A. K. (2002). Joining multiple data streams with window constraints.

Hammad, M. A., Aref, W. G., & Elmagarmid, A. K. (2008). Query processing of multi-way stream window joins. The VLDB Journal—The International Journal on Very Large Data Bases, 17(3), 469-488.

Hammad, M. A., Franklin, M. J., Aref, W. G., & Elmagarmid, A. K. (2003, September). Scheduling for shared window joins over data streams. *In Proceedings of the 29th international conference on Very large data bases-Volume 29* (pp. 297-308). VLDB Endowment.

Heinz, C., Kramer, J., Riemenschneider, T., & Seeger, B. (2008, April). Toward simulation-based optimization in data stream management systems. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on* (pp. 1580-1583). IEEE.

Heinze, T., Jerzak, Z., Hackenbroich, G., & Fetzer, C. (2014, May). Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (pp. 13-22). ACM.

Kang, J., Naughton, J. F., & Viglas, S. D. (2003, March). Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on* (pp. 341-352). IEEE.

Kossmann, D., & Stocker, K. (2000). Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)*, *25*(1), 43-82.

Lohman, G. M. (2014, April). Is query optimization a'solved'problem. In *Proc. Workshop on Database Query Optimization* (p. 13). Oregon Graduate Center Comp. Sci. Tech. Rep.

Madden, S., Shah, M., Hellerstein, J. M., & Raman, V. (2002, June). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (pp. 49-60). ACM.

Park, H. K., & Lee, W. S. (2009). Adaptive continuous query reoptimization over data streams. *IEICE transactions on information and systems*, *92*(7), 1421-1428.
Park, H. K., & Lee, W. S. (2012). Adaptive optimization for multiple continuous queries. *Data & Knowledge Engineering*, *71*(1), 29-46.

Roy, P., Seshadri, S., Sudarshan, S., & Bhobe, S. (2000). Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, *29*(2), 249-260.

Rundensteiner, E. A., Ding, L., Sutherland, T., Zhu, Y., Pielech, B., & Mehta, N. (2004, August). Cape: Continuous query engine with heterogeneous-grained adaptivity. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (pp. 1353-1356). VLDB Endowment.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979, May). Access path selection in a relational database management system. *In Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (pp. 23-34). ACM.

Sellis, T., & Ghosh, S. (1990). On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), 262-266.

Sirish, C., Owen, C., Amol, D., Wei, H., Sailesh, K., Samuel, M., ... & Frederick, R. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. *CIDR'03*.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., & Stonebraker, M. (2003, September). Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29* (pp. 309-320). VLDB Endowment.

Tatbul, N., & Zdonik, S. (2006, September). Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases* (pp. 799-810). VLDB Endowment.

To, Q. C., Soto, J., & Markl, V. (2017). A Survey of State Management in Big Data Processing Systems. *arXiv preprint arXiv:1702.01596.*

Urhan, T., & Franklin, M. J. (2000). Xjoin: A reactively-scheduled pipelined join operatorý. *Bulletin of the Technical Committee on*, 27.

Viglas, S. D., & Naughton, J. F. (2002, June). Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (pp. 37-48). ACM.

Viglas, S. D., Naughton, J. F., & Burger, J. (2003, September). Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases-Volume 29* (pp. 285-296). VLDB Endowment.

Viglas, S. D. (2005). Query Execution and Optimization. In *Stream Data Management* (pp. 15-33). Springer US.

Wilschut, A. N., & Apers, P. M. (1990). Pipelining in Query Executions.

Zhu, Y., Raghavan, V., & Rundensteiner, E. A. (2010). A new look at generating multi-join continuous query plans: A qualified plan generation problem. *Data & Knowledge Engineering*, *69*(5), 424-443.

APPENDIX A

A.1 Query Order Experiment

5 join predicates	For the data stream: 1: The Input Rate is:29 For the data stream: 2: The Input Rate is:40 For the data stream: 3: The Input Rate is:18 For the data stream: 4: The Input Rate is:34 For the data stream: 5: The Input Rate is:91 For the data stream: 6: The Input Rate is:79 For the data stream: 7: The Input Rate is:43 For the data stream: 8: The Input Rate is:84 For the data stream: 9: The Input Rate is:60 For the data stream: 10: The Input Rate is:43 For the data stream: 11: The Input Rate is:57 For the data stream: 12: The Input Rate is:75	For (1,2), the selectivity factor is:0.11 For (2,3), the selectivity factor is:0.19 For (1,4), the selectivity factor is:0.15 For (4,5), the selectivity factor is:0.19 For (5,6), the selectivity factor is:0.03 For (4,7), the selectivity factor is:0.06 For (7,8), the selectivity factor is:0.16 For (5,9), the selectivity factor is:0.14 For (9,10), the selectivity factor is:0.12 For (4,11), the selectivity factor is:0.19 For (9,12), the selectivity factor is:0.11

A.2 Resource Consumption Without Constraints on CPU or memory

3 join predicates	For the data stream: 1: The Input Rate is:69 For the data stream: 2: The Input Rate is:28 For the data stream: 3: The Input Rate is:31 For the data stream: 4: The Input Rate is:26 For the data stream: 5: The Input Rate is:96 For the data stream: 6: The Input Rate is:60 For the data stream: 7:	For (1,2), the selectivity factor is:0.2 For (2,3), the selectivity factor is:0.09 For (2,4), the selectivity factor is:0.13 For (4,5), the selectivity factor is:0.09 For (5,6), the selectivity factor is:0.17 For (4,7), the selectivity factor is:0.18 For (7,8), the selectivity

	The Input Rate is:42 For the data stream: 8: The Input Rate is:50 For the data stream: 9: The Input Rate is:30 For the data stream: 10: The Input Rate is:32 For the data stream: 11: The Input Rate is:61 For the data stream: 12: The Input Rate is:11 For the data stream: 13: The Input Rate is:17 For the data stream: 14: The Input Rate is:90 For the data stream: 15: The Input Rate is:96 For the data stream: 16: The Input Rate is:36 For the data stream: 17: The Input Rate is:83 For the data stream: 18: The Input Rate is:28 For the data stream: 19: The Input Rate is:44 For the data stream: 20: The Input Rate is:12	factor is:0.09 For (6,9), the selectivity factor is:0.2 For (6,10), the selectivity factor is:0.1 For (6,11), the selectivity factor is:0.17 For (9,12), the selectivity factor is:0.19 For (11,13), the selectivity factor is:0.2 For (12,14), the selectivity factor is:0.16 For (5,15), the selectivity factor is:0.08 For (10,16), the selectivity factor is:0.04 For (10,17), the selectivity factor is:0.18 For (1,18), the selectivity factor is:0.12 For (3,19), the selectivity factor is:0.02 For (14,20), the selectivity factor is:0.13
5 join predicates	For the data stream: 1: The Input Rate is:30 For the data stream: 2: The Input Rate is:77 For the data stream: 3: The Input Rate is:73 For the data stream: 4: The Input Rate is:19 For the data stream: 5: The Input Rate is:94 For the data stream: 6: The Input Rate is:100 For the data stream: 7: The Input Rate is:86 For the data stream: 8: The Input Rate is:23 For the data stream: 9: The Input Rate is:71	For (1,2), the selectivity factor is:0.01 For (1,3), the selectivity factor is:0.05 For (1,4), the selectivity factor is:0.04 For (1,5), the selectivity factor is:0.02 For (3,6), the selectivity factor is:0.01 For (4,7), the selectivity factor is:0.2 For (3,8), the selectivity factor is:0.15 For (8,9), the selectivity factor is:0.06 For (9,10), the selectivity factor is:0.14

	For the data stream: 10: The Input Rate is:11	For (7,11), the selectivity factor is:0.08
	For the data stream: 11: The Input Rate is:52	For (5,12), the selectivity factor is:0.15
	For the data stream: 12: The Input Rate is:80	For (8,13), the selectivity factor is:0.14
	For the data stream: 13: The Input Rate is:64	For (9,14), the selectivity factor is:0.12
	For the data stream: 14: The Input Rate is:65	For (4,15), the selectivity factor is:0.16
	For the data stream: 15: The Input Rate is:15	For (7,16), the selectivity factor is:0.15
	For the data stream: 16: The Input Rate is:31	For (15,17), the selectivity factor is:0.13
	For the data stream: 17: The Input Rate is:32	For (9,18), the selectivity factor is:0.17
	For the data stream: 18: The Input Rate is:40	For (10,19), the selectivity factor is:0.17
	For the data stream: 19: The Input Rate is:89	For (14,20), the selectivity factor is:0.18
	For the data stream: 20: The Input Rate is:77	
7 join predicates	For the data stream: 1: The Input Rate is:78	For (1,2), the selectivity factor is:0.13
	For the data stream: 2: The Input Rate is:33	For (1,3), the selectivity factor is:0.18
	For the data stream: 3: The Input Rate is:15	For (3,4), the selectivity factor is:0.03
	For the data stream: 4: The Input Rate is:89	For (1,5), the selectivity factor is:0.11
	For the data stream: 5: The Input Rate is:58	For (4,6), the selectivity factor is:0.17
	For the data stream: 6: The Input Rate is:63	For (4,7), the selectivity factor is:0.17
	For the data stream: 7: The Input Rate is:46	For (3,8), the selectivity factor is:0.17
	For the data stream: 8: The Input Rate is:53	For (6,9), the selectivity factor is:0.03
	For the data stream: 9: The Input Rate is:78	For (2,10), the selectivity factor is:0.01
	For the data stream: 10: The Input Rate is:91	For (3,11), the selectivity factor is:0.03
	For the data stream: 11: The Input Rate is:97	For (9,12), the selectivity factor is:0.17
	For the data stream: 12:	For (2,13), the selectivity

	The Input Rate is:81 For the data stream: 13: The Input Rate is:29 For the data stream: 14: The Input Rate is:40 For the data stream: 15: The Input Rate is:94 For the data stream: 16: The Input Rate is:58 For the data stream: 17: The Input Rate is:50 For the data stream: 18: The Input Rate is:39 For the data stream: 19: The Input Rate is:30 For the data stream: 20: The Input Rate is:89	factor is:0.19 For (7,14), the selectivity factor is:0.11 For (14,15), the selectivity factor is:0.18 For (8,16), the selectivity factor is:0.17 For (13,17), the selectivity factor is:0.02 For (1,18), the selectivity factor is:0.01 For (14,19), the selectivity factor is:0.17 For (19,20), the selectivity factor is:0.05
9 join predicate	For the data stream: 1: The Input Rate is:84 For the data stream: 2: The Input Rate is:37 For the data stream: 3: The Input Rate is:57 For the data stream: 4: The Input Rate is:68 For the data stream: 5: The Input Rate is:33 For the data stream: 6: The Input Rate is:87 For the data stream: 7: The Input Rate is:13 For the data stream: 8: The Input Rate is:63 For the data stream: 9: The Input Rate is:76 For the data stream: 10: The Input Rate is:17 For the data stream: 11: The Input Rate is:44 For the data stream: 12: The Input Rate is:23 For the data stream: 13: The Input Rate is:77 For the data stream: 14: The Input Rate is:87	For (1,2), the selectivity factor is:0.04 For (2,3), the selectivity factor is:0.03 For (3,4), the selectivity factor is:0.11 For (2,5), the selectivity factor is:0.16 For (5,6), the selectivity factor is:0.17 For (3,7), the selectivity factor is:0.05 For (1,8), the selectivity factor is:0.04 For (1,9), the selectivity factor is:0.14 For (8,10), the selectivity factor is:0.07 For (1,11), the selectivity factor is:0.1 For (8,12), the selectivity factor is:0.01 For (5,13), the selectivity factor is:0.1 For (1,14), the selectivity factor is:0.01 For (3,15), the selectivity factor is:0.06

	For the data stream: 15: The Input Rate is:82 For the data stream: 16: The Input Rate is:81 For the data stream: 17: The Input Rate is:43 For the data stream: 18: The Input Rate is:76 For the data stream: 19: The Input Rate is:48 For the data stream: 20: The Input Rate is:53	For (8,16), the selectivity factor is:0.17 For (2,17), the selectivity factor is:0.09 For (6,18), the selectivity factor is:0.01 For (12,19), the selectivity factor is:0.12 For (12,20), the selectivity factor is:0.08

A.3 Resource Consumption With Constraints on both CPU and memory

3 join predicates	For the data stream: 1: The Input Rate is:50 For the data stream: 2: The Input Rate is:84 For the data stream: 3: The Input Rate is:21 For the data stream: 4: The Input Rate is:89 For the data stream: 5: The Input Rate is:41 For the data stream: 6: The Input Rate is:34 For the data stream: 7: The Input Rate is:65 For the data stream: 8: The Input Rate is:61 For the data stream: 9: The Input Rate is:39 For the data stream: 10: The Input Rate is:18 For the data stream: 11: The Input Rate is:67 For the data stream: 12: The Input Rate is:47 For the data stream: 13: The Input Rate is:83 For the data stream: 14: The Input Rate is:68 For the data stream: 15: The Input Rate is:38	For (1,2), the selectivity factor is:0.16 For (2,3), the selectivity factor is:0.2 For (1,4), the selectivity factor is:0.18 For (4,5), the selectivity factor is:0.2 For (5,6), the selectivity factor is:0.02 For (1,7), the selectivity factor is:0.11 For (1,8), the selectivity factor is:0.16 For (6,9), the selectivity factor is:0.07 For (9,10), the selectivity factor is:0.14 For (1,11), the selectivity factor is:0.09 For (8,12), the selectivity factor is:0.02 For (6,13), the selectivity factor is:0.03 For (12,14), the selectivity factor is:0.05 For (3,15), the selectivity factor is:0.01 For (6,16), the selectivity factor is:0.05

	For the data stream: 16: The Input Rate is:58 For the data stream: 17: The Input Rate is:48 For the data stream: 18: The Input Rate is:50 For the data stream: 19: The Input Rate is:82 For the data stream: 20: The Input Rate is:91	For (14,17), the selectivity factor is:0.01 For (17,18), the selectivity factor is:0.11 For (12,19), the selectivity factor is:0.16 For (5,20), the selectivity factor is:0.08
5 join predicates	For the data stream: 1: The Input Rate is:15 For the data stream: 2: The Input Rate is:57 For the data stream: 3: The Input Rate is:38 For the data stream: 4: The Input Rate is:23 For the data stream: 5: The Input Rate is:90 For the data stream: 6: The Input Rate is:61 For the data stream: 7: The Input Rate is:95 For the data stream: 8: The Input Rate is:51 For the data stream: 9: The Input Rate is:54 For the data stream: 10: The Input Rate is:36 For the data stream: 11: The Input Rate is:15 For the data stream: 12: The Input Rate is:26 For the data stream: 13: The Input Rate is:90 For the data stream: 14: The Input Rate is:97 For the data stream: 15: The Input Rate is:57 For the data stream: 16: The Input Rate is:43 For the data stream: 17: The Input Rate is:32 For the data stream: 18:	For (1,2), the selectivity factor is:0.01 For (2,3), the selectivity factor is:0.11 For (1,4), the selectivity factor is:0.18 For (3,5), the selectivity factor is:0.08 For (2,6), the selectivity factor is:0.19 For (6,7), the selectivity factor is:0.09 For (7,8), the selectivity factor is:0.02 For (6,9), the selectivity factor is:0.11 For (7,10), the selectivity factor is:0.06 For (8,11), the selectivity factor is:0.05 For (11,12), the selectivity factor is:0.18 For (6,13), the selectivity factor is:0.04 For (7,14), the selectivity factor is:0.11 For (7,15), the selectivity factor is:0.03 For (5,16), the selectivity factor is:0.09 For (15,17), the selectivity factor is:0.19 For (13,18), the selectivity factor is:0.17 For (8,19), the selectivity

	The Input Rate is:92 For the data stream: 19: The Input Rate is:15 For the data stream: 20: The Input Rate is:45	factor is:0.11 For (2,20), the selectivity factor is:0.04
7 join predicates	For the data stream: 1: The Input Rate is:10 For the data stream: 2: The Input Rate is:42 For the data stream: 3: The Input Rate is:33 For the data stream: 4: The Input Rate is:52 For the data stream: 5: The Input Rate is:19 For the data stream: 6: The Input Rate is:31 For the data stream: 7: The Input Rate is:76 For the data stream: 8: The Input Rate is:78 For the data stream: 9: The Input Rate is:77 For the data stream: 10: The Input Rate is:51 For the data stream: 11: The Input Rate is:86 For the data stream: 12: The Input Rate is:60 For the data stream: 13: The Input Rate is:96 For the data stream: 14: The Input Rate is:42 For the data stream: 15: The Input Rate is:28 For the data stream: 16: The Input Rate is:20 For the data stream: 17: The Input Rate is:61 For the data stream: 18: The Input Rate is:50 For the data stream: 19: The Input Rate is:98 For the data stream: 20: The Input Rate is:72	For (1,2), the selectivity factor is:0.15 For (1,3), the selectivity factor is:0.12 For (1,4), the selectivity factor is:0.06 For (3,5), the selectivity factor is:0.11 For (5,6), the selectivity factor is:0.1 For (5,7), the selectivity factor is:0.09 For (2,8), the selectivity factor is:0.18 For (2,9), the selectivity factor is:0.15 For (2,10), the selectivity factor is:0.17 For (10,11), the selectivity factor is:0.03 For (8,12), the selectivity factor is:0.17 For (12,13), the selectivity factor is:0.11 For (3,14), the selectivity factor is:0.06 For (5,15), the selectivity factor is:0.05 For (2,16), the selectivity factor is:0.2 For (12,17), the selectivity factor is:0.1 For (16,18), the selectivity factor is:0.16 For (14,19), the selectivity factor is:0.1 For (12,20), the selectivity factor is:0.18

9 join predicates	For the data stream: 1: The Input Rate is:16 For the data stream: 2: The Input Rate is:18 For the data stream: 3: The Input Rate is:16 For the data stream: 4: The Input Rate is:23 For the data stream: 5: The Input Rate is:85 For the data stream: 6: The Input Rate is:16 For the data stream: 7: The Input Rate is:22 For the data stream: 8: The Input Rate is:44 For the data stream: 9: The Input Rate is:34 For the data stream: 10: The Input Rate is:65 For the data stream: 11: The Input Rate is:76 For the data stream: 12: The Input Rate is:85 For the data stream: 13: The Input Rate is:86 For the data stream: 14: The Input Rate is:57 For the data stream: 15: The Input Rate is:68 For the data stream: 16: The Input Rate is:86 For the data stream: 17: The Input Rate is:62 For the data stream: 18: The Input Rate is:78 For the data stream: 19: The Input Rate is:72 For the data stream: 20: The Input Rate is:46	For (1,2), the selectivity factor is:0.08 For (1,3), the selectivity factor is:0.15 For (1,4), the selectivity factor is:0.16 For (2,5), the selectivity factor is:0.13 For (4,6), the selectivity factor is:0.09 For (5,7), the selectivity factor is:0.02 For (4,8), the selectivity factor is:0.1 For (8,9), the selectivity factor is:0.12 For (4,10), the selectivity factor is:0.14 For (4,11), the selectivity factor is:0.06 For (11,12), the selectivity factor is:0.12 For (1,13), the selectivity factor is:0.04 For (1,14), the selectivity factor is:0.17 For (6,15), the selectivity factor is:0.09 For (9,16), the selectivity factor is:0.18 For (16,17), the selectivity factor is:0.16 For (11,18), the selectivity factor is:0.05 For (4,19), the selectivity factor is:0.12 For (5,20), the selectivity factor is:0.12

A.4 CPU and Memory Consumption When Restricting Memory Resources

3 join predicates	For the data stream: 1: The Input Rate is:12	For (1,2), the selectivity factor is:0.01

	For the data stream: 2: The Input Rate is:96 For the data stream: 3: The Input Rate is:25 For the data stream: 4: The Input Rate is:74 For the data stream: 5: The Input Rate is:74 For the data stream: 6: The Input Rate is:44 For the data stream: 7: The Input Rate is:27 For the data stream: 8: The Input Rate is:71 For the data stream: 9: The Input Rate is:30 For the data stream: 10: The Input Rate is:39 For the data stream: 11: The Input Rate is:13 For the data stream: 12: The Input Rate is:22 For the data stream: 13: The Input Rate is:42 For the data stream: 14: The Input Rate is:81 For the data stream: 15: The Input Rate is:32 For the data stream: 16: The Input Rate is:57 For the data stream: 17: The Input Rate is:29 For the data stream: 18: The Input Rate is:23 For the data stream: 19: The Input Rate is:49 For the data stream: 20: The Input Rate is:30	For (1,3), the selectivity factor is:0.08 For (2,4), the selectivity factor is:0.15 For (1,5), the selectivity factor is:0.02 For (2,6), the selectivity factor is:0.11 For (4,7), the selectivity factor is:0.09 For (1,8), the selectivity factor is:0.12 For (4,9), the selectivity factor is:0.18 For (1,10), the selectivity factor is:0.06 For (7,11), the selectivity factor is:0.05 For (2,12), the selectivity factor is:0.11 For (1,13), the selectivity factor is:0.02 For (6,14), the selectivity factor is:0.18 For (6,15), the selectivity factor is:0.17 For (3,16), the selectivity factor is:0.02 For (1,17), the selectivity factor is:0.02 For (4,18), the selectivity factor is:0.19 For (8,19), the selectivity factor is:0.13 For (18,20), the selectivity factor is:0.06
5 join predicates	For the data stream: 1: The Input Rate is:92 For the data stream: 2: The Input Rate is:23 For the data stream: 3: The Input Rate is:71 For the data stream: 4:	For (1,2), the selectivity factor is:0.16 For (2,3), the selectivity factor is:0.15 For (2,4), the selectivity factor is:0.07 For (2,5), the selectivity

	The Input Rate is:18 For the data stream: 5: The Input Rate is:50 For the data stream: 6: The Input Rate is:29 For the data stream: 7: The Input Rate is:46 For the data stream: 8: The Input Rate is:57 For the data stream: 9: The Input Rate is:98 For the data stream: 10: The Input Rate is:99 For the data stream: 11: The Input Rate is:80 For the data stream: 12: The Input Rate is:50 For the data stream: 13: The Input Rate is:79 For the data stream: 14: The Input Rate is:78 For the data stream: 15: The Input Rate is:56 For the data stream: 16: The Input Rate is:51 For the data stream: 17: The Input Rate is:30 For the data stream: 18: The Input Rate is:72 For the data stream: 19: The Input Rate is:12 For the data stream: 20: The Input Rate is:17	factor is:0.13 For (3,6), the selectivity factor is:0.16 For (5,7), the selectivity factor is:0.07 For (5,8), the selectivity factor is:0.01 For (8,9), the selectivity factor is:0.02 For (6,10), the selectivity factor is:0.07 For (3,11), the selectivity factor is:0.2 For (5,12), the selectivity factor is:0.04 For (3,13), the selectivity factor is:0.08 For (3,14), the selectivity factor is:0.04 For (4,15), the selectivity factor is:0.04 For (7,16), the selectivity factor is:0.13 For (5,17), the selectivity factor is:0.14 For (4,18), the selectivity factor is:0.17 For (5,19), the selectivity factor is:0.02 For (16,20), the selectivity factor is:0.07
7 join predicates	For the data stream: 1: The Input Rate is:43 For the data stream: 2: The Input Rate is:41 For the data stream: 3: The Input Rate is:32 For the data stream: 4: The Input Rate is:27 For the data stream: 5: The Input Rate is:77 For the data stream: 6: The Input Rate is:24	For (1,2), the selectivity factor is:0.17 For (2,3), the selectivity factor is:0.16 For (2,4), the selectivity factor is:0.04 For (2,5), the selectivity factor is:0.19 For (3,6), the selectivity factor is:0.11 For (1,7), the selectivity factor is:0.04

	For the data stream: 7: The Input Rate is:76 For the data stream: 8: The Input Rate is:14 For the data stream: 9: The Input Rate is:95 For the data stream: 10: The Input Rate is:77 For the data stream: 11: The Input Rate is:35 For the data stream: 12: The Input Rate is:53 For the data stream: 13: The Input Rate is:92 For the data stream: 14: The Input Rate is:13 For the data stream: 15: The Input Rate is:100 For the data stream: 16: The Input Rate is:54 For the data stream: 17: The Input Rate is:34 For the data stream: 18: The Input Rate is:58 For the data stream: 19: The Input Rate is:49 For the data stream: 20: The Input Rate is:33	For (3,8), the selectivity factor is:0.12 For (7,9), the selectivity factor is:0.1 For (4,10), the selectivity factor is:0.04 For (4,11), the selectivity factor is:0.03 For (5,12), the selectivity factor is:0.02 For (4,13), the selectivity factor is:0.11 For (3,14), the selectivity factor is:0.05 For (6,15), the selectivity factor is:0.13 For (9,16), the selectivity factor is:0.19 For (11,17), the selectivity factor is:0.09 For (17,18), the selectivity factor is:0.06 For (13,19), the selectivity factor is:0.03 For (12,20), the selectivity factor is:0.14
9 join predicates	For the data stream: 1: The Input Rate is:46 For the data stream: 2: The Input Rate is:14 For the data stream: 3: The Input Rate is:22 For the data stream: 4: The Input Rate is:31 For the data stream: 5: The Input Rate is:35 For the data stream: 6: The Input Rate is:80 For the data stream: 7: The Input Rate is:48 For the data stream: 8: The Input Rate is:52 For the data stream: 9:	For (1,2), the selectivity factor is:0.02 For (2,3), the selectivity factor is:0.06 For (3,4), the selectivity factor is:0.02 For (2,5), the selectivity factor is:0.01 For (1,6), the selectivity factor is:0.1 For (6,7), the selectivity factor is:0.16 For (6,8), the selectivity factor is:0.04 For (1,9), the selectivity factor is:0.07 For (8,10), the selectivity

	The Input Rate is:80 For the data stream: 10: The Input Rate is:52 For the data stream: 11: The Input Rate is:19 For the data stream: 12: The Input Rate is:49 For the data stream: 13: The Input Rate is:85 For the data stream: 14: The Input Rate is:54 For the data stream: 15: The Input Rate is:42 For the data stream: 16: The Input Rate is:99 For the data stream: 17: The Input Rate is:71 For the data stream: 18: The Input Rate is:94 For the data stream: 19: The Input Rate is:40 For the data stream: 20: The Input Rate is:15	factor is:0.13 For (5,11), the selectivity factor is:0.15 For (3,12), the selectivity factor is:0.09 For (3,13), the selectivity factor is:0.01 For (6,14), the selectivity factor is:0.12 For (8,15), the selectivity factor is:0.02 For (7,16), the selectivity factor is:0.11 For (7,17), the selectivity factor is:0.11 For (2,18), the selectivity factor is:0.15 For (3,19), the selectivity factor is:0.08 For (8,20), the selectivity factor is:0.01

A.5 CPU and Memory Consumption When Restricting CPU Resources

3 join predicates	For the data stream: 1: The Input Rate is:93 For the data stream: 2: The Input Rate is:14 For the data stream: 3: The Input Rate is:70 For the data stream: 4: The Input Rate is:48 For the data stream: 5: The Input Rate is:62 For the data stream: 6: The Input Rate is:11 For the data stream: 7: The Input Rate is:72 For the data stream: 8: The Input Rate is:25 For the data stream: 9: The Input Rate is:66	For (1,2), the selectivity factor is:0.18 For (1,3), the selectivity factor is:0.18 For (1,4), the selectivity factor is:0.01 For (1,5), the selectivity factor is:0.19 For (4,6), the selectivity factor is:0.1 For (6,7), the selectivity factor is:0.16 For (6,8), the selectivity factor is:0.16 For (2,9), the selectivity factor is:0.03 For (6,10), the selectivity factor is:0.2

	For the data stream: 10: The Input Rate is:58	For (9,11), the selectivity factor is:0.16
	For the data stream: 11: The Input Rate is:81	For (9,12), the selectivity factor is:0.08
	For the data stream: 12: The Input Rate is:32	For (7,13), the selectivity factor is:0.01
	For the data stream: 13: The Input Rate is:19	For (4,14), the selectivity factor is:0.17
	For the data stream: 14: The Input Rate is:65	For (10,15), the selectivity factor is:0.05
	For the data stream: 15: The Input Rate is:80	For (10,16), the selectivity factor is:0.02
	For the data stream: 16: The Input Rate is:84	For (15,17), the selectivity factor is:0.01
	For the data stream: 17: The Input Rate is:87	For (15,18), the selectivity factor is:0.08
	For the data stream: 18: The Input Rate is:42	For (4,19), the selectivity factor is:0.11
	For the data stream: 19: The Input Rate is:95	For (14,20), the selectivity factor is:0.17
	For the data stream: 20: The Input Rate is:28	
5 join predicates	For the data stream: 1: The Input Rate is:95	For (1,2), the selectivity factor is:0.12
	For the data stream: 2: The Input Rate is:59	For (2,3), the selectivity factor is:0.12
	For the data stream: 3: The Input Rate is:41	For (2,4), the selectivity factor is:0.03
	For the data stream: 4: The Input Rate is:33	For (2,5), the selectivity factor is:0.17
	For the data stream: 5: The Input Rate is:19	For (2,6), the selectivity factor is:0.18
	For the data stream: 6: The Input Rate is:48	For (5,7), the selectivity factor is:0.13
	For the data stream: 7: The Input Rate is:90	For (3,8), the selectivity factor is:0.04
	For the data stream: 8: The Input Rate is:58	For (5,9), the selectivity factor is:0.05
	For the data stream: 9: The Input Rate is:15	For (3,10), the selectivity factor is:0.2
	For the data stream: 10: The Input Rate is:99	For (3,11), the selectivity factor is:0.07
	For the data stream: 11: The Input Rate is:88	For (3,12), the selectivity factor is:0.17
	For the data stream: 12:	For (2,13), the selectivity

	The Input Rate is:99	factor is:0.13
	For the data stream: 13:	For (6,14), the selectivity
	The Input Rate is:10	factor is:0.17
	For the data stream: 14:	For (4,15), the selectivity
	The Input Rate is:18	factor is:0.04
	For the data stream: 15:	For (1,16), the selectivity
	The Input Rate is:78	factor is:0.01
	For the data stream: 16:	For (2,17), the selectivity
	The Input Rate is:99	factor is:0.16
	For the data stream: 17:	For (9,18), the selectivity
	The Input Rate is:23	factor is:0.18
	For the data stream: 18:	For (17,19), the selectivity
	The Input Rate is:65	factor is:0.01
	For the data stream: 19:	For (10,20), the selectivity
	The Input Rate is:83	factor is:0.08
	For the data stream: 20:	
	The Input Rate is:26	
7 join predicates	For the data stream: 1:	For (1,2), the selectivity
	The Input Rate is:80	factor is:0.14
	For the data stream: 2:	For (1,3), the selectivity
	The Input Rate is:28	factor is:0.12
	For the data stream: 3:	For (2,4), the selectivity
	The Input Rate is:94	factor is:0.02
	For the data stream: 4:	For (3,5), the selectivity
	The Input Rate is:51	factor is:0.01
	For the data stream: 5:	For (2,6), the selectivity
	The Input Rate is:11	factor is:0.1
	For the data stream: 6:	For (3,7), the selectivity
	The Input Rate is:92	factor is:0.15
	For the data stream: 7:	For (5,8), the selectivity
	The Input Rate is:14	factor is:0.01
	For the data stream: 8:	For (8,9), the selectivity
	The Input Rate is:31	factor is:0.2
	For the data stream: 9:	For (4,10), the selectivity
	The Input Rate is:50	factor is:0.09
	For the data stream: 10:	For (7,11), the selectivity
	The Input Rate is:37	factor is:0.09
	For the data stream: 11:	For (9,12), the selectivity
	The Input Rate is:56	factor is:0.1
	For the data stream: 12:	For (5,13), the selectivity
	The Input Rate is:80	factor is:0.19
	For the data stream: 13:	For (10,14), the selectivity
	The Input Rate is:31	factor is:0.05
	For the data stream: 14:	For (3,15), the selectivity
	The Input Rate is:67	factor is:0.18

	For the data stream: 15: The Input Rate is:42	For (10,16), the selectivity factor is:0.05
	For the data stream: 16: The Input Rate is:53	For (1,17), the selectivity factor is:0.03
	For the data stream: 17: The Input Rate is:53	For (4,18), the selectivity factor is:0.13
	For the data stream: 18: The Input Rate is:29	For (16,19), the selectivity factor is:0.2
	For the data stream: 19: The Input Rate is:58	For (14,20), the selectivity factor is:0.03
	For the data stream: 20: The Input Rate is:15	
9 join predicates	For the data stream: 1: The Input Rate is:99	For (1,2), the selectivity factor is:0.18
	For the data stream: 2: The Input Rate is:12	For (2,3), the selectivity factor is:0.02
	For the data stream: 3: The Input Rate is:31	For (3,4), the selectivity factor is:0.2
	For the data stream: 4: The Input Rate is:20	For (4,5), the selectivity factor is:0.15
	For the data stream: 5: The Input Rate is:57	For (1,6), the selectivity factor is:0.07
	For the data stream: 6: The Input Rate is:85	For (4,7), the selectivity factor is:0.03
	For the data stream: 7: The Input Rate is:39	For (7,8), the selectivity factor is:0.07
	For the data stream: 8: The Input Rate is:48	For (5,9), the selectivity factor is:0.03
	For the data stream: 9: The Input Rate is:65	For (2,10), the selectivity factor is:0.2
	For the data stream: 10: The Input Rate is:98	For (3,11), the selectivity factor is:0.13
	For the data stream: 11: The Input Rate is:58	For (10,12), the selectivity factor is:0.02
	For the data stream: 12: The Input Rate is:32	For (2,13), the selectivity factor is:0.04
	For the data stream: 13: The Input Rate is:71	For (12,14), the selectivity factor is:0.06
	For the data stream: 14: The Input Rate is:34	For (1,15), the selectivity factor is:0.1
	For the data stream: 15: The Input Rate is:60	For (12,16), the selectivity factor is:0.18
	For the data stream: 16: The Input Rate is:24	For (9,17), the selectivity factor is:0.19
	For the data stream: 17:	For (1,18), the selectivity

	The Input Rate is:90 For the data stream: 18: The Input Rate is:15 For the data stream: 19: The Input Rate is:60 For the data stream: 20: The Input Rate is:17	factor is:0.01 For (11,19), the selectivity factor is:0.17 For (13,20), the selectivity factor is:0.04