Multiple Continuous Query Processing with Relative Window Predicates

by

Asima Silva

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

April 2004

APPROVED:

Professor Elke Rundensteiner, Thesis Advisor

Professor George Heineman, Thesis Advisor

Professor David Finkel, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

Efficient querying over streaming data is a critical technology which requires the ability to handle numerous and possibly similar queries in real time dynamic environments such as the stock market and medical devices. Existing DBMS technology is not well suited for this domain since it was developed for static historical data. Queries over streams often contain relative window predicates such as in the query: "Heart rate decreased to fifty-two beats per second within four seconds after the patient's temperature started rising." Relative window predicates are a specific type of join between streams that is based on the tuple's timestamp.

In our operator, called Juggler, predicates are classified into three types: attribute, join, and window. Attribute predicates are stream values compared to a constant. Join predicates are stream values compared to another stream's values. Window predicates are join predicates where the streams' timestamp values are compared. Juggler's composite operator incorporates the processing of similar though not identical, query functionalities as one complex computation process. This execution strategy handles multi-way joins for multiple selection and join predicates. It adaptively orders the execution of predicates by their selectivity to efficiently process multiple continuous queries based on stream characteristics. In Juggler, all similar predicates are grouped into lists. These indices are represented by a collection of bits. Every tuple contains the bit structure representation of the predicate lists which encodes tuple predicate evaluation history. Every query also contains a similar bit structure to encode the predicate's relationship to the registered queries. The tuple's and query's bit structures are compared to assess if the tuple has satisfied a query.

Juggler is designed and implemented in Java. Experiments were conducted to verify correctness and to assess the performance of Juggler's three features. Its adaptivity of reordering the evaluation of predicate types performed as well as the most selective predicate ordering. Its ability to exploit similar predicates in multiple queries showed reduction in number of comparisons. Its effectiveness when multiple queries are combined in a single Juggler operator indicated potential performance improvements after optimization of Juggler's data structures.

Acknowledgements

I would like to thank my advisors, Professor Elke Rundensteiner and Professor George Heineman. They have both given me guidance and support throughout my graduate program. I would also like to thank my thesis reader, Professor David Finkel, for his patience and feedback. Lastly, I would like to thank my family, my husband, David Silva, and my children, Mudassir, Zainab, and Maimoona for their understanding and support during my long hours and tight schedule.

Most importantly, I would like to dedicate my work to my grandfather, Hameeduddin Ahmed. His passion for knowledge was inspiring. His support, confidence, and encouragement have been the cornerstone of my academic achievements. His sacrifices to educate his family remain a timeless gift.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Continuous Queries vs. Traditional Databases 2

1.3 Problem Definition 3

1.4 State of the Art 5

1.5 Juggler Overview 5

1.6 Outline 8

2 Related Work **10**

3 Juggler Architecture **19**

3.1 Query Representation 19

3.1.1 Queries in Running Example 21

3.2 Sharing Computation 25

3.2.1 Predicate Lists 27

3.2.2 Juggler Predicate BitSet Structure 30

3.2.3 Query Encoding Dependency Structure 32

3.2.4 Predicate BitSet Structure 34

3.3 Storing Tuples in Operator 36

3.3.1 Join Exploitation Structures 37

3.3.2 Window Exploitation Structure 38
3.4 Adaptive Predicate Ordering 41
3.4.1 Tuple's JugglerPBS 41
3.4.2 Predicate Ordering 42
3.4.3 Output Tuples 44
3.4.4 Insert Tuples into Exploitation Structures 45
3.4.5 Clean Up 46
3.4.6 Juggler Architecture Overview 47

4 Running Example **49**
4.1 Optimal Selective Predicate Ordering 50
4.1.1 First Predicate Type Evaluation 50
4.2 Second Predicate Type Evaluation 55
4.3 Third Predicate Type Evaluation 60
4.4 Output Tuples 61
4.5 Clean Up 63
4.6 Operator's Processing Logic 65

5 Assessing Juggler **67**
5.1 Cost Model 67
5.1.1 Selectivity 67
5.1.2 Space Cost 80
5.2 Experimental Evaluation 84
5.2.1 Binary versus Multi-joins 85
5.2.2 Predicate Reordering 88
5.2.3 Overlapping Predicates 90

6 Conclusion **96**

6.1 Future Work 98

6.1.1 Adaptive Predicate Reordering 98

6.1.2 Policies 98

6.1.3 Optimization 99

6.1.4 Performance 99

A Juggler Implementation **104**

A.1 Juggler Operator 104

A.2 Predicate Ordering Manager 106

A.3 Predicate Type Manager 106

A.3.1 Attribute Predicate Manager 107

A.3.2 Join Predicate Manager 108

A.3.3 Window Predicate Manager 108

A.4 Exploitation Structures 109

A.4.1 Join Exploitation Structure 110

A.4.2 Window Exploitation Structure 111

A.5 Predicate Structure 112

A.5.1 Attribute Predicate 113

A.5.2 Join and Window Predicates 114

A.6 Predicate List 114

A.6.1 Attribute Predicate List 115

A.7 Join/Window Predicate List 115

A.8 Predicate Lists 116

A.8.1 Attribute Predicate Lists 117

A.8.2 Join/Window Predicate Lists 117

A.8.3 BitPosition 117

A.9 BitSet Collection 118
A.9.1 General Predicate BitSet 118
A.9.2 Predicate BitSet 119
A.9.3 ExtGeneralType BitSet 119
A.9.4 ExtPredicate BitSet 119
A.10 Juggler Predicate BitSet Structure 120
A.10.1 Relevant Predicate BitSet Structure 120
A.10.2 Satisfied Predicate BitSet Structure 120
A.11 Query Encoding Dependency 121
A.11.1 Juggler Operator's QEDs 121
A.12 Juggler Tuple 122
A.12.1 JugglerTupleList 122
A.12.2 JugglerWindowTupleList 123
A.13 Juggler Comparative Keys 124
A.14 Juggler Comparative Operators 124

List of Figures

1.1 CAPE Architecture 6

1.2 Juggler vs. Traditional Binary Operators in a Query Plan 8

2.1 Current CQ Research Topics 11

3.1 Juggler Query Representation 20

3.2 Juggler Query Representation Example 21

3.3 Queries in Juggler 22

3.4 Query plans as Complex DAGs 23

3.5 One Possible Query Plan with Juggler Operators 24

3.6 Query plan with Juggler Operator 25

3.7 Calculating Maximum Window Size for Another Possible Query plan 26

3.8 Calculating Maximum Window Size for a Query plan 27

3.9 Algorithm for Registering Predicates in Juggler 27

3.10 Attribute Predicate Lists and BitPositions 28

3.11 Join Predicate Lists and BitPositions 28

3.12 Window Predicate Lists and BitPositions 28

3.13 QEDs for node BTH 32

3.14 Juggler Tuple's Relevant Predicate BitSet Structure (RelPBS) 34

3.15 Bit Comparisons with QEDs and tuple's JugglerPBS 35

3.16 Tuples In Juggler Operator BTHK 36
3.17 Join Exploitation Structures (JESs) in BTHK 37
3.18 Window Exploitation Structures for Juggler Operator BTHK 39
3.19 Join Algorithm 43
3.20 Tuple Output Algorithm 44
3.21 Inserting Tuples into Exploitation Structures Algorithm 45
3.22 Juggler Clean Up Algorithm 46
3.23 Logical Architecture of Juggler Operator 48

4.1 Initial Juggler Predicate BitSet Structure (JugglerPBS) 50
4.2 JugglerPBSbefore First Predicate Type Evaluation in the WAJ Predicate Ordering 50
4.3 First Window Bucket of Stream B 50
4.4 Intermediate Tuples After First Predicate Type Evaluation in WAJ Predicate Ordering 53
4.5 Intermediate Joined Tuples' Query List After First Predicate Type Evaluation in WAJ Predicate Ordering 55
4.6 Tuples' RelPBS for Second Predicate Type in WAJ Predicate Ordering 56
4.7 Find Most Covering Predicate Algorithm 57
4.8 Tuples after attribute predicate type evaluation 58
4.9 Tuples' Query List after Second Predicate Type 60
4.10 Tuples' RelPBS before Third Predicate Type Evaluation for WAJ Predicate Ordering 60
4.11 Tuples after Third Predicate Type Evaluation in WAJ ordering . . . 61
4.12 WES after processing input tuples 64
4.13 WES after purging stale tuples 65
4.14 Architecture of an Operator 66

4.15 Juggler Operator's Functionality 66

5.1 Queries used to Compare Cost of Overlapping Predicates 76
5.2 Query Plans with Binary Operators 85
5.3 Query Plans with Multi-join Operators 85
5.4 Binary and Multi-join Query Plans 86
5.5 Number of Intermediate Tuples for Binary and Multi-join Query Plans 87
5.6 Tuple Overhead for Binary and Multi-join Query Plans 88
5.7 Number of Comparisons for Binary and Multi-join Query Plans . . . 89
5.8 Query used to test Predicate Reordering 89
5.9 Query Plans with One Multi-join Operator 90
5.10 Adaptive vs. Static Predicate Orderings 91
5.11 Number of Comparisons for Predicate Orderings 92
5.12 Queries with Overlapping Predicates 92
5.13 Query 1 Predicate Overlap 93
5.14 Query 2 Predicate Overlap 94
5.15 Query 3 Predicate Overlap 95

A.1 Juggler Operator Interface 125
A.2 Predicate Type Managers 126
A.3 Predicate Exploitation Structures 127
A.4 Predicates 128
A.5 Predicate List Structures 129
A.6 Predicate BitSet Structures 130
A.7 Query Structures 131
A.8 Juggler Tuple Structures 132
A.9 Juggler Comparative Keys 133

A.10 Juggler Comparative Operators 134

Chapter 1

Introduction

1.1 Motivation

Continuous queries (CQ) are continuously evaluated queries over streaming data. They are found in many domains that process real-time data such as financial systems, network management, and medical monitoring. The ability to process and query streaming data from multiple devices can be a powerful technology. For example, critical patients in the intensive care unit (ICU) are constantly monitored with instruments, such as heart monitors, IV drips, oxygen machines, and heart/kidney pumps. Currently, data gathered from these machines is displayed for the nurses to monitor a patient's condition. There is no system today that allows doctors and nurses to specify queries that monitor a patient's condition as it evaluates the data gathered. Traditional databases only handle static stored data not dynamic data streams.

1.2 Continuous Queries vs. Traditional Databases

Recently, several proposed systems, including STREAM [3] and Rate Based Query Optimization [17], have addressed CQ challenges. In order to handle changing stream statistics, dynamically adaptive routing and join algorithms have been explored. Reducing computation is another key requirement for streaming data. Our goal is to investigate continuous queries with sliding window joins within this wide area of CQ requirements.

Continuous queries are an emerging new research area with many unexplored issues outlined in *Models and Issues in Data Stream Systems*[2] and *Adaptive Query Processing: Technology in Evolution* [8]. Some recent research in continuous queries include NiagaraCQ [5], Ripple Join [6], Window joins [9], XJoin [14], and MJoin [15].

Continuous queries differ in many ways from traditional databases which process queries over static, typically persistently stored data. Before a query is executed in a traditional database, an optimizer creates a query plan which orders join and select operations based on known data characteristics [13]. In the traditional approach, queries are only run once, when a user executes a query. Results are output after a query completes. This approach also assumes that the data usually changes only through infrequent updates. Therefore, a snapshot of the data is kept and synchronized to reflect the updates.

Continuous queries are computed over streaming data, generally within the lifespan of a specified window. Factors such as stream statistics and stream data rates are thus ever-changing in a continuous query environment. Considering the nature of continuous data, a complete result is impossible because the data stream may be infinite. Similarly, continuous queries differ from the traditional approach regarding

storage requirements and memory management.

These issues are traditionally handled off-line whereas continuous queries cannot afford to expend resources on static design strategies. One such design is the traditional memory management technique of storing data on disks. Infinite storage, required for continuous queries, is unrealistic. Another issue is query plan optimization. While a static query plan can be generated based on known statistics in a continuous query environment, a dynamic query plan should constantly be revised to reflect the dynamically changing stream statistics. Lastly, in a traditional system, the output is returned only after it has been fully computed, whereas continuous output of intermediate results is necessary in many real-time environments. A continuous query system must address the above issues related to adaptivity.

1.3 Problem Definition

Continuous queries requires processing of data in a real-time streaming environment. Juggler proposes a solution with three contributions.

- Grouping similar attribute and join predicates.
- Reordering predicate evaluation using bit structures to maintain history.
- A multi-join operator that processes multiple joins and selects.

Grouping attribute predicates have been investigated by XML subscriptions [12]. Join predicates have been only grouped to share evaluation if identical. Juggler, on the other hand, groups similar attribute and join predicates to evaluate concurrently in order to share sub-computations.

Reordering predicate evaluation in an operator also has not been done. Changing tuple path of evaluation [1] or dynamic query plan migration has been investigated.

Introducing an operator that reorders predicate evaluation is a novel idea. Juggler groups similar predicates and reorders evaluation to adapt to changing stream statistics.

Traditionally, operators have a single functionality. If the operator's functionality is to process an attribute predicate, it is a single input operator. If the operator's functionality is to process a join predicate, it is a binary input operator. Streaming environments are usually characterized by a limited number of streams with an unlimited number of continuous queries over these streams. For this reason, MJoin [16] has investigated a multi-join operator in this environment. Combining multiple predicates in one operator has not been proposed. In CQ, the probability of queries containing similar predicates is significant. Juggler exploits this characteristic by proposing a multi-join operator that evaluates multiple predicates.

Even though some of Juggler's contributions have been proposed, the combination of these features in one operator is novel. Each contribution was tested to assess Juggler's performance. As the number of similar predicates increased, the number of comparisons significantly decreased. Also, the reordering of predicate evaluation within the operator adapted to changing data stream distributions. This is a key feature necessary in CQ environments. Lastly, the performance of multi-joins and its equivalent binary join operators were compared. Juggler's multi-join feature displayed a comparable output rate, but had a significant reduction in the number of comparisons. Overall, Juggler's features promise further performance improvements with optimizations.

1.4 State of the Art

Our system, Juggler, incorporates innovative ideas from Eddies [1], SteMs [10], and M-join [16]. Juggler uses complex DAGs to represent all possible query plans. It also incorporates Eddies' [1] idea of dynamically choosing query paths based on statistics. Relying on their findings that using bits to encode information does not incur substantial overhead, Juggler encodes intermediate processing information as bits associated with each tuple. Juggler also aims to confirm and extend M-join's [16] findings that one multi-join in many cases is more efficient than its equivalent binary joins for streaming environments. Lastly, the idea of sharing joins with similar predicates has been used in XML subscriptions [12]. Juggler introduces a complex operator for computing several predicates which are similar or even overlapping. Not only are all these ideas combined into a system, but Juggler now proposes a novel adaptive predicate ordering scheme that is more suitable for a dynamically changing environment.

1.5 Juggler Overview

Juggler is a multi-join operator which tackles the problem of continuously querying over streaming data in real-time. Juggler offers a solution which incorporates multiple query plans, joins, and selects into one composite operator. It bounds the streaming data using window joins and a window size. To enable the adaptive evaluation of predicates, Juggler categorizes the predicates into three types: attribute, join, and window. Attribute predicates are filter expressions that compare one attribute of a stream value to a constant value. Join predicates are binary expressions that compare one attribute value of a stream against the value of another stream. Window predicates are a specific type of join predicate in which the streams' times-

tamp values are compared. These types are dynamically reordered and applied in order of selectivity, which is the number of tuples output by a predicate type evaluation divided by the Cartesian product these accounts for the potential size. Juggler can dynamically adapt to the changing stream data distribution. These issues have been investigated by several systems, yet Juggler's approach of dynamically ordering predicate types in a multi-join mega operator is novel.

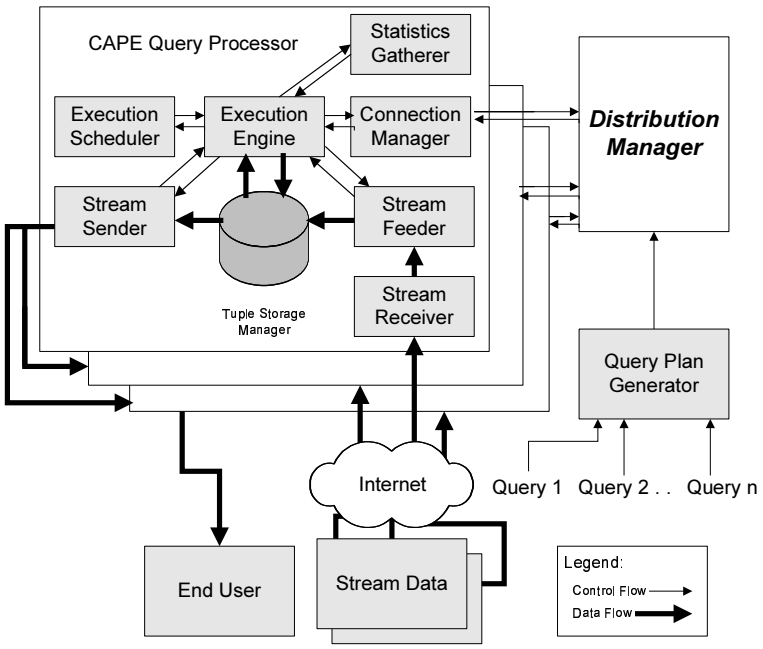

Figure 1.1: CAPE Architecture

The Juggler operator runs within the CAPE system, a Continuous Adaptive Processing Engine implemented at WPI by members of the Database Systems Research Group, Figure 1.1. The query plans were defined and entered into the Query Plan Generator. CAPE's ExecutionController calls operators in the system and runs them for a specified amount of time. Currently, the query plans used for this thesis' experimentation only contained Juggler operators. In the future, Juggler operators can be replaced with other operator(s) with different implementations to compare

performance.

Juggler handles multiple continuous queries with theta joins and relative window predicates. It categorizes predicates into three groups: attribute, join and window. Juggler adaptively reorders the evaluation of these groups to allow for adaptiveness to dynamic stream characteristics. Juggler also handles multiple streams and numerous continuous queries.

Given initial input stream statistics, such as data value ranges and arrival rates, a query plan can be designed and entered into the CAPE system. The query plan contains a combination of traditional binary join operators and Juggler multi-join operators. When there are similar predicates in one query or even several, Juggler combines these predicates into groups in order to evaluate multiple predicates at time to reduce the number of comparisons.

CAPE processes queries by distributing the evaluation over several operators. CAPE uses a StreamGenerator which streams data at a predefined rate. It also uses an ExecutionController which creates operators input queues and output queues. It also runs the operators in a query plan for a specified time. During CAPE's initialization, CAPE's config.xml file is parsed to create operators and initialize each.

Juggler merges these registered queries into one global query plan, if possible, grouping query predicates such that similar predicates can be combined into one composite operator, as shown in Figure 1.2. The goal of this thesis was to design and implement such a composite operator, which we will call Juggler. Three types of predicates will be considered: attribute, join and window. These predicate types will be reordered to reflect changes in stream data distributions.

Juggler has three contributions in the CQ environment. First, it groups similar predicates and dynamically reorders the order of predicate type evaluation. Sec-

ondly, it uses bit structures to encode tuple evaluation history and correlate the relationship of predicates to queries, and vice versa. Third, it is a multi-join operator that can process multiple streams incorporating joins and selects into one mega operator. The combination of these three features results in reducing number of comparisons when queries have many overlapping or similar predicates.

The Juggler composite operator joins, applies predicates, and projects the joined tuples to its parent nodes in the query plan. I have concentrated on the composite operator design, its cost model, and the adaptive predicate ordering. I have designed and developed the Juggler operator within the CAPE system implemented with Java 1.4.

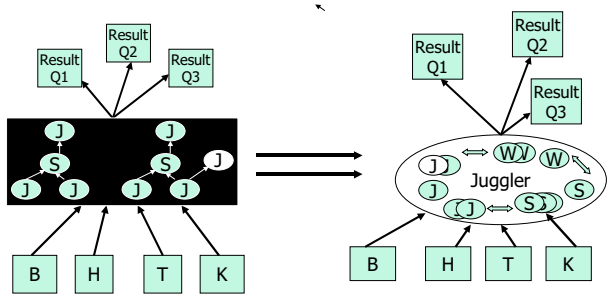

Figure 1.2: Juggler vs. Traditional Binary Operators in a Query Plan

1.6 Outline

The rest of this thesis will be organized as follows. Chapter 2 will describe research in the CQ area and how each compares to Juggler. Chapter 3 describes the core

structures that Juggler utilizes. Juggler's query representation and Predicate BitSets are two key structures. Chapter 3 also describes how Juggler maintains the tuple predicate evaluation history, storage of these tuples in the operator and its join algorithm. Chapter 4 describes Juggler's operator using a running example. The process of the evaluation of one tuple is followed in detail. Chapter 5 details Juggler's cost model and describes experiments that were conducted. Conclusions are derived from the experimental results in Chapter 6, and tasks are outlined for future work. Appendix A contains implementation details of Juggler's join algorithm and data structures.

Chapter 2

Related Work

Our operator, Juggler, incorporates innovative ideas from Eddies [1], SteMs [10], and M-join [16]. Much research has been done in adaptive query processing, maximizing output rate, and handling multiple continuous queries with multiple streaming data streams. Three research topics that have been explored are shown in Figure 2.1: routing algorithms, join algorithms, and exploring different semantics of bounding streaming data using windows. Many proposed solutions have tried to address one or more of these issues.

Adaptivity is a core issue for continuous queries. With ever-changing stream data and its characteristics, static solutions are not viable. Current research has spanned the issues outlined above. Eddies [1], SteMs [10], and Ripple Join[6] address both the routing and join algorithms in the CQ environment. CACQ [11], MJoin [16], WJoin [7], and PSoup [4] have addressed window semantics, routing, and join algorithms.

Eddies [1] proposes dynamic reordering of operators. In this single query system, each tuple follows its own customized order of visiting operators. Eddies routes tuples to available operators and this availability is determined by a lottery scheme. This scheme utilizes the operator's queue size and output rate to determine the

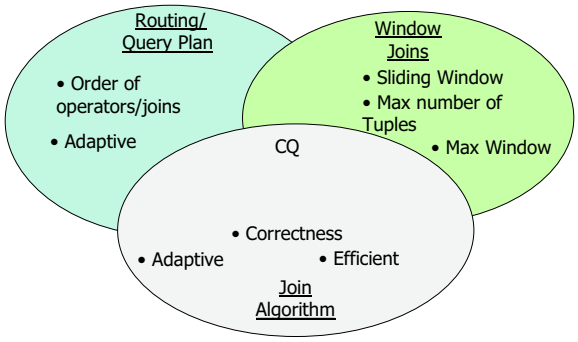

Figure 2.1: Current CQ Research Topics

tuple's routing path. With this scheme, the tuple is routed to operators that are available rather than contribute more work to an already over-loaded operator. An operator's availability is kept track of by tickets. When a tuple enters an operator, the ticket count is incremented, and when tuples are output, tickets are decremented. An operator with a small number of tickets implies it is not overwhelmed. In this way, the Eddies algorithm adapts to variations in stream selectivity by dynamically reordering operator evaluation.

To maintain a tuple's evaluation history, Eddies uses a bit mechanism. Two sets of bits are used to maintain evaluated and unevaluated operators to be processed, named *done* and *ready* respectively. Each bit represents operators in the system. The setting of a tuple's *ready* bit indicates that the operators represented by the bit still remains to be evaluated. A setting of a tuple's *done* bit indicates the operators have been evaluated. Eddies' experimentation and evaluation indicate that bits do not add significant overhead. Therefore Eddies represents an efficient continuous query routing system.

While Eddies handles a single query requiring a pre-optimizer to determine operators in the system, STeMs [10], based on Eddies, requires no pre-optimizer and calculates everything on the fly. This query processing algorithm is based on half-join state modules. Using Eddies as its routing algorithm, SteMs proposes an improved join algorithm, which contains half-join operators. This allows sharing of intermediate evaluations. It incorporates selections, joins, and adaptive query plans all in one operator. Each SteM groups predicates by operator, reducing the number of tuple comparisons when evaluating attribute predicates. These predicates compare a stream's value to a constant. Its basic algorithm builds tables of tuples grouped by streams that serve as a cache or a hash table. When a tuple enters the system, its value is used to find matching tuples to join from the hash tables. Global timestamps are assigned to each tuple to avoid duplicates.

Both STeMs and Eddies use the dynamic Ripple Join [6] algorithm. This binary join algorithm exploits the ordering of its inputs. When one data input arrives, it is joined with the other input. If one input becomes blocked, the other input's data is used to join. This algorithm adapts to varying arrival rates of the streams, appropriate for CQ environments.

Our operator, Juggler, applies similar ideas to the Eddies concept of dynamic reordering. Juggler is a multi-join operator that processes multiple queries and reorders predicate type evaluations. Eddies is a single query system, whereas Juggler handles multiple query processing. Eddies uses bits to encode processing and routing information. Like Eddies, Juggler uses bits to encode relevant and satisfied predicates within the operator. These bits determine the predicates that remain to be evaluated for a given tuple, and thus, determine its path within the Juggler operator. Also similar to Eddies's Ripple Join, Juggler's join algorithm updates the operator's predicate type selectivities and reorders the predicate types, thereby

handling variations in stream selectivities.

STeMs is similar to Juggler in that it shares intermediate predicate evaluations and uses bits to maintain the evaluation history. Juggler is a composite operator that combines joins and selects into one operator, whereas, SteMs is a half join operator that processes only attribute predicates. Juggler groups both attribute and join predicates to reduce comparisons. SteMs only groups attribute predicates. Also SteMs is composed of single stream and binary stream operators whereas, Juggler processes multiple join, selects and multiple queries in one operator.

STeMs shares some similarities with Juggler. The Juggler operator aims to incorporate selections, joins, and an adaptive query plan all into one mega operator. Both STeMs and Juggler share the idea of building structures of tuples, a temporary cache, which allows efficient access to retrieve relevant tuples.

CACQ [11], MJoin [16], WJoin [7], and PSoup [4] have investigated combining several current CQ research topics: window semantics, routing, and join algorithms. Juggler also investigated these topics. These proposed systems aim to accomplish one or more CQ goals: maximizing output rate, reducing computation, and finding dynamic, adaptive, and scalable solutions.

CACQ [11], based on both STeMs and Eddies, is a Continuously Adaptive Continuous Query system. It handles multiple continuous queries by grouping filters for selections. Joins are split into SteMs, a half-join operator, allowing sharing of joins between multiple queries. Eddies is used as the tuple router, choosing a path at the granularity of each tuple. Building on Eddies' maintenance of tuple evaluation history, CACQ also uses bits to maintain a list of queries the tuple has satisfied. Each tuple is appended with *done* and *ready* bits, as in Eddies, and *queriesCompleted* bits. The *queriesCompleted* bits indicate which queries have been satisfied. Before a tuple is output, the *queriesCompleted* bits are compared to the indicated queries'

CompletionMask bits. CACQ allows operators to be shared by multiple queries by using two more bit structures, *queriesCompleted* and *CompletionMask*, than SteMs. Similar to Eddies, CACQ maintains that the bit structures do not add substantial overhead.

CACQ addresses similar ideas, however, instead of incorporating adaptive query plans into one single operator as done in Juggler, it requires a router that chooses dynamic paths. Juggler incorporates joins and selects into one operator, but CACQ relies on multiple half join state modules to process tuples, SteMs.

SteMs is very similar to Juggler's attribute predicate type since it groups attribute predicates and evaluates multiple selections at a time. CACQ requires a dynamic router and STeMs in order to achieve what Juggler attempts to accomplish with its one operator. Juggler not only combines attribute predicates, it also groups join and window predicates. This could further reduce the number of intermediate tuples in the system and can also further reduce the number of comparisons. These two strategies can be compared in the future.

M-join [16] proposes a multi-way symmetric join operator which implements binary operators in one single functionality. It considers sliding window where tuples outside of a defined time window are considered stale and no longer joined or processed. It aims to produce outputs sooner than its equivalent binary operators. Performance is based on rate rather than cardinality. It claims to adapt to changing rates of input streams. A multi-way join also reduces the need to modify the query plan in order to adapt to its changing environment, since a multi-way join incorporates several possible query plans. It builds on concepts of XJoin, flushing tuples to disk for processing at a later time. Tuples are partitioned by timestamps, and stored in hash tables. Partitions are used to retrieve tuples within a query's window, and all possible joined tuples within the window is then filtered by the

other remaining predicates before being output. MJoin recommends at most five inputs for the multi-way join to avoid degradation in performance [16].

W-join [7] introduces several join algorithms for a continuous query with a sliding window. All streams are assumed to be involved in a single query containing one join predicate. Three join algorithms, all variations of W-join, are described and compared: nested-loop (NLW-join), hash (HW-join), and merge join (MW-join). MW-join consists of two modes. The first mode identifies the relevant window for a tuple to be joined. The second mode admits tuples that are contained in the relevant window and joins them. Three different window constraints are considered: a single maximum window over a query, relative window constraints between streams, and no window constraints on a stream. NLW-join does not perform well in situations when a stream blocks. HW-join assumes all the join predicates are equality joins. MW-join outperformed NLW-join and behaves well even under variable stream rates.

PSoup [4], based on the Telegraph project, extends the concept of Eddies and SteMs. It processes multiple queries by creating indexes on both queries and data. All predicates in the system are assumed to be attribute predicates which compare a stream's value to a constant. Queries can be swapped in and out of the system by creating and removing query indexes. These indices allow similar processing for an entry of a new tuple and a new query. PSoup incrementally maintains a materialized view of joined tuples within a window size in order to quickly respond to a user's intermittent request for output. The joined state is shared by the queries in the system. Also, window size is defined by the number of tuples and not timestamp range.

Juggler offers a key adaptive feature that MJoin does not. MJoin is a multi-join operator that combines all possible query plans, but it does not exploit changing stream statistics. MJoin's join algorithm traverses each of its input streams to create

a joined tuples from all streams within a time window. After all the possible joined output tuples are created, MJoin filters these tuples using the predicates registered in its operator. Juggler, on the other hand, is not only a multi-join operator which incorporates multiple query plans, it also filters tuples earlier in the algorithm by reordering predicate types by selectivity. This reduces the number of intermediate tuples which in turn reduces the number of comparisons.

W-join only considers a single continuous query containing one join predicate. Juggler introduces an adaptive join algorithm which processes multiple continuous queries with multiple predicates in one operator. The queries registered in Juggler are not restricted to a specific number of predicates and consist of multiple selections and joins.

PSoup addresses registering and deregistering continuous queries into its system dynamically. It processes tuples in real-time, but only outputs in response to user requests. PSoup only contains queries with attribute predicates, while Juggler handles both join and window predicates. Juggler does not consider query registration and deregistration. It leaves this task as future work.

Juggler is similar in many ways to the work described above. Juggler is a multi-way join operator that evaluates queries composed of attribute and join predicates. These predicates include equality and theta joins evaluated over a sliding window.

Juggler introduces an adaptive join algorithm by categorizing predicates into three types. These predicate types are reordered to adapt to changing stream statistics. Each predicate type is evaluated in order of selectivity, reducing the number of intermediate tuples. Only relevant predicates are applied and used to retrieve tuples to join. These temporary joined tuples contain a list of query IDs that the tuple has satisfied during the evaluation process. This list is updated after each evaluation phase. After the third and last predicate evaluation phase, the tuple's list of query

IDs represent the queries that the tuple has satisfied.

Juggler's join algorithm exploits the overlap of predicates in multiple queries. It aims to reduce computation by applying the query's most selective predicate type first. Unlike PSoup where a user request determines when a result is output, Juggler outputs results incrementally, otherwise known as sliding window semantics. It does not process any results over historical data larger than the operator's window size, as PSoup and M-Join do. Maximum window size of all the queries determines if data has become stale. Juggler does not store any tuples that lie outside of the maximum window to process at a later time. Many applications, such as the medical monitoring devices, have no value for historical results. PSoup is suited for an environment that is triggered by frequency of user requests.

Juggler represents a balance between the Eddies routing scheme and M-join. Eddies sends individual tuples across different paths in a possibly huge query plan. Juggler utilizes selectivities of predicate types to route tuples through the appropriate paths in the operator. Like Eddies, Juggler provides a evaluation of tuples that adapts to the changing selectivities of the predicate types. Traditional binary joins output more tuples per operator whereas the equivalent multi-way join used in Juggler and M-join reduces this number. The combination of a dynamic routing scheme and multi-way joins can lead to performance improvements.

Juggler's operator runs within the CAPE system and uses DAGs, directed acyclic graphs, to represent all possible query plans. Each path chosen in a DAG represents one possible query plan. Relying on Eddies' findings that using bits to encode information does not incur substantial overhead, Juggler encodes intermediate processing information as bits associated with each tuple. Juggler also aims to confirm and extend M-join's [16] findings that one multi-join, in many cases, is more efficient than its equivalent binary joins for streaming environments. Lastly, the idea of shar-

ing similar predicates has been used in XML subscriptions [12] but only attribute predicates have been considered. Predicates are grouped and ordered such that predicates are only evaluated if dependent or more covering predicates are satisfied. Juggler also aims to extend sharing sub computations to join predicates.

Juggler introduces a complex operator for computing several similar or even overlapping predicates. These predicates consist of single stream selects and complex joins, which include theta operators such as $A.col1 >= B.col1$. Not only are all these ideas combined into a system, but Juggler now proposes a novel adaptive predicate ordering scheme that is suitable for a highly dynamically changing environment.

Chapter 3

Juggler Architecture

Juggler is composed of several structures that enable it to process tuples and share sub computations. Structures that aid in sharing sub computations between queries include: Predicate Lists, Predicate BitSets, and Query Encoding Dependency. Other structures are used to store and quickly retrieve tuples to join, such as the Join Exploitation Structure and the Window Exploitation Structure. These core structures are used in Juggler's adaptive join algorithm to reduce number of comparisons and filter tuples at an earlier stage of processing.

3.1 Query Representation

Queries are registered into the CAPE system using the syntax shown in Figure 3.1. The *Select*, *From*, and *Where* clauses are as in SQL. The *From* clause defines the streams involved in the query. The *Where* clause contains attribute and join predicates. These predicates compare a stream's value to a constant or to another stream's value, respectively. *Window* and *Max Window* clauses are used specifically by Juggler. The *Window* clause are time oriented constraints between query's streams. These are specific type of theta joins based on stream timestamps. *Max*

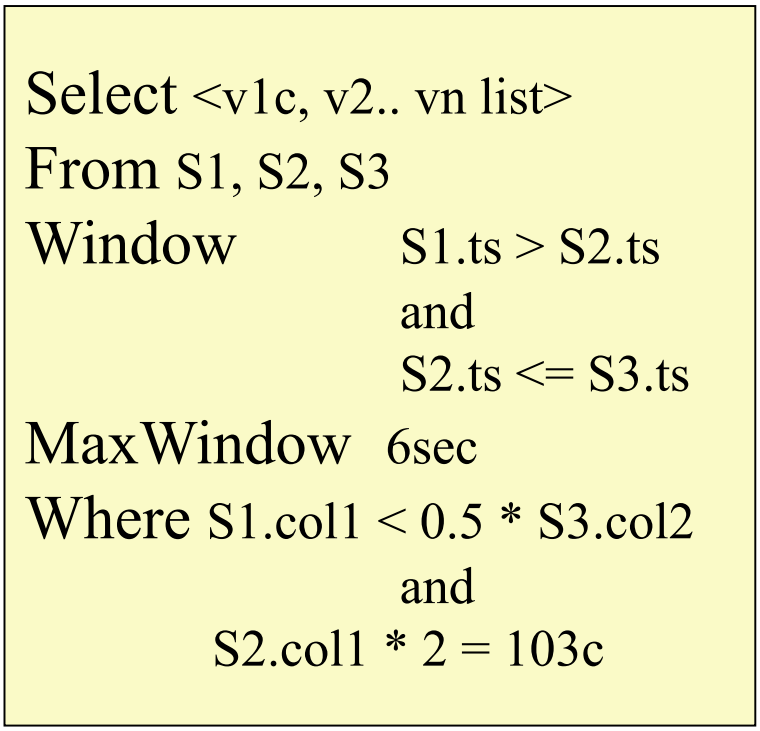

Figure 3.1: Juggler Query Representation

Window clauses defines the maximum window or maximum window range of the query. This window size is a time range that shifts as time passes, called a sliding window. Only data within this window size is considered when processing a tuple. Any data outside this window size is considered stale and irrelevant. Queries are evaluated as each input tuple is being processed in the operator.

For example, the query shown in Figure 3.2 involves three streams: blood, temp and heart. It contains one attribute predicate, *T.fluct * 2 = 103*, and one join predicate, *T.fluct < 0.5 * B.pressure*. The *Window* clause defines the query's relative window predicates, *B.ts > T.ts* and *B.ts <= H.ts*. Relative window predicates are either theta or equi-joins on the streams' timestamps. Lastly, the *Max Window* clause bounds the data to a window size of six seconds which is evaluated in a sliding fashion. All queries registered in Juggler are specified in this query language and

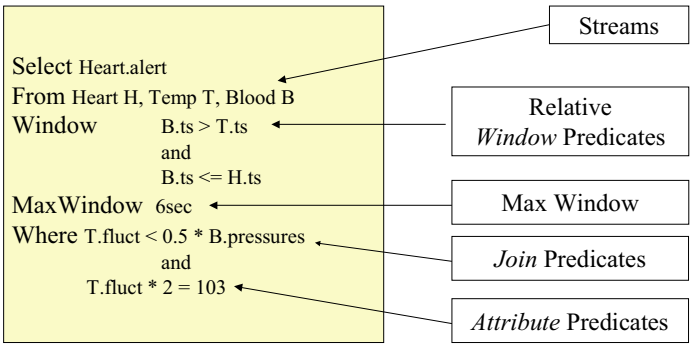

Figure 3.2: Juggler Query Representation Example

are given a query ID.

3.1.1 Queries in Running Example

Many scenarios can occur with intensive care unit (ICU) patients. Patients are constantly monitored by machines and nurses. Each machine indicates a critical condition based on values of its data streams. Nurses are responsible for monitoring each data stream and correlating possible dangers. Using Juggler, each scenario can be monitored automatically. Figure 3.3 describes queries registered in Juggler, assuming four data streams: blood (B), temperature (T), kidney (K), and heart (H). Each query describes a possible critical scenario. All queries define the set of streams considered, join predicates, attribute predicates, and window predicates, relative and general.

In Juggler, queries that contain at least one predicate for each type will use Juggler's join algorithm to its fullest. If a query does not contain a predicate type,

Query ID: 1

Select	H.alert
From	B, H, K, T
Window	T.ts > B.ts
MaxWindow	4sec
Where	H.beatrate = 52 and
	B.temp > H.vib and
	B.pressure > T.fluct

Query ID: 2

Select	B.alert
From	B, T, H
Window	B.ts > T.ts and
	B.ts <= H.ts
MaxWindow	6sec
Where	T.fluct < 0.5 * B.pressure
	T.fluct * 2 = 103

Query ID: 3

Select	H.alert
From	T, H
Window	T.ts > H.ts
MaxWindow	5sec
Where	T.fluct = 103 and
	T.incr = H.vib

Figure 3.3: Queries in Juggler

the filtering process after the corresponding predicate type evaluation is missed. This reduces the chance to filter tuples as early as possible, which increases the number of intermediate tuples kept in the operator while processing the input tuples.

Query 1, for example, is a join of blood, temperature, kidney, and heart (streams BTKH). Query 2 is a join of blood, temperature, and heart (streams BTH) and Query 3 is a join of temperature and heart (streams TH). Note, even though the queries do not contain identical combination of the operator's input streams, Juggler can group them into one operator.

Traditionally, each query to be executed can be represented by a query plan composed of primitive operators as shown in Figure 3.4. Query plans are used to order operations of joins and selects. Juggler's goal is to combine plans, subsets of possibly several query plans into one mega-operator. This is accomplished by grouping similar predicates, collecting statistics at run-time, and evaluating predicate types

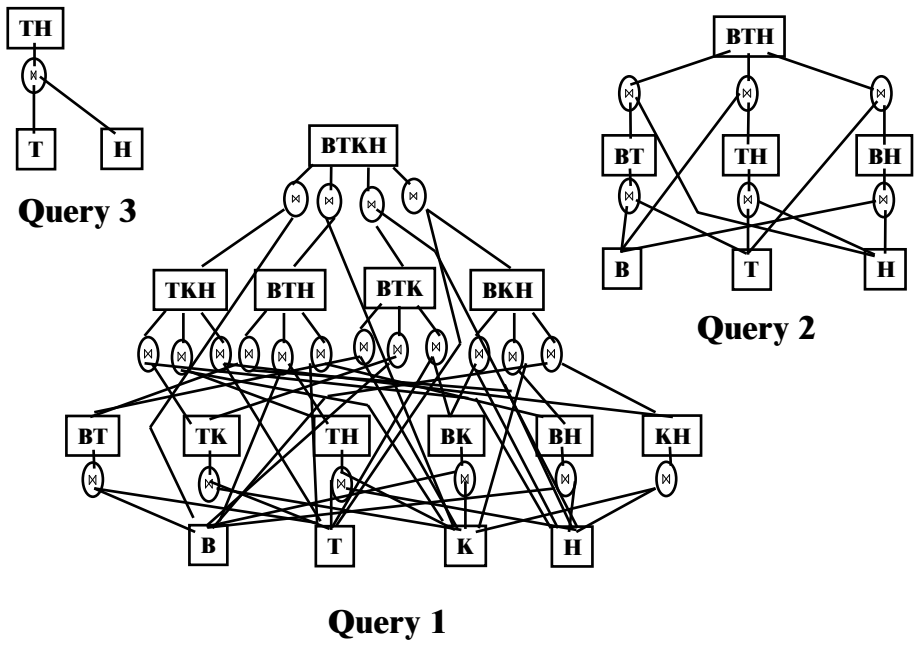

Figure 3.4: Query plans as Complex DAGs

in order of selectivity.

Each query can be represented by a query plan as seen in Figure 3.4. Each operator represents a sub computation, which can be either a join or select. The joined tuple output from an operator is represented by combining the names of its inputs. For example, a node BT computes the join of blood and temperature streams. Note that each query plan contains many alternative paths for achieving the same query semantics. Two possible query plans with Juggler operators are shown in Figures 3.5 and 3.6. A specific query plan will be selected by the constructed by the optimizer based on cost estimates. In Figure 3.6 for example Queries 1, 2, and 3 share the Juggler operator, BTHK.

When a Juggler operator is shared with multiple queries, the window size of the operator must be computed. We use two query plans for the queries defined

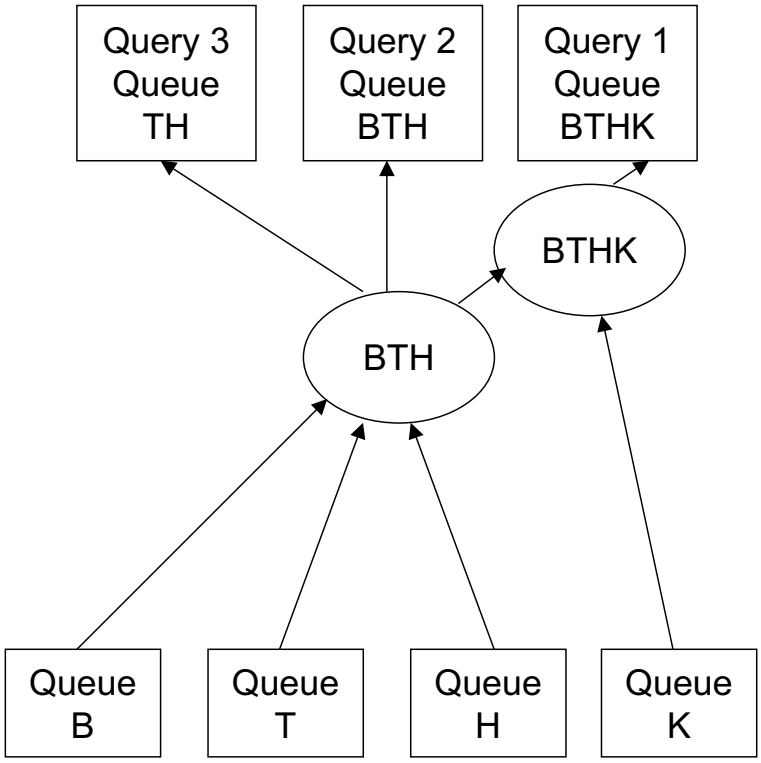

Figure 3.5: One Possible Query Plan with Juggler Operators

in Figure 3.3 to display the calculation of each operators' window size. These two query plans are shown in Figures 3.7 and 3.8. The largest of the Maximum Window sizes of the registered queries in an operator will become the operator's maximum window size. Taking the largest window size of the queries allows sharing of sub computation satisfying all the queries' window sizes. The query plan and maximum window size shown in Figure 3.7 will be used in our running example.

BTHK is a multi-join operator that evaluates predicates for all three queries. An operator will only contain predicates for queries that are contained in the operator and predicates that involve its input streams. For example, in Figure 3.5, operators BTH and BTHK are Juggler operators. In this case, the operator BTH will contain predicates that involve only the streams B, T, and H. Similarly, the operator BHTK will only contain predicates that involve only the streams B, T, H, and H. It will also

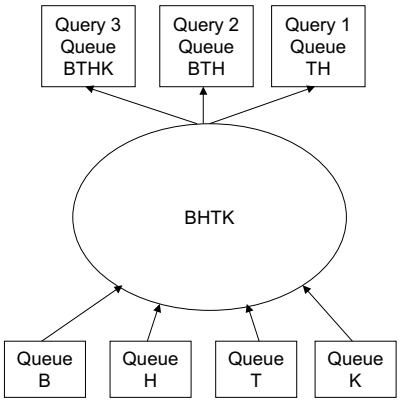

Figure 3.6: Query plan with Juggler Operator

contain predicates that have not been evaluated by operator BTH. For example, if Query 1 had a predicate, $B.pressure > K.fluid$, which could not have been evaluated by operator BTH, operator BTHK would evaluate this predicate before any tuples are output for Query 1. As Query 1 is defined in our example, since the all the predicates can be evaluated and processed by BHT, BHTK would only evaluate if tuples K are within the maximum window range of tuples BHT.

3.2 Sharing Computation

Juggler's operator combines the query plans of the registered queries to share sub computations. It also dynamically reacts to changing stream characteristics by reordering predicate type evaluation. Juggler's responsibility can be summarized in three requirements.

- First, Juggler needs a mechanism to share computation and maintain tuple predicate evaluation history.

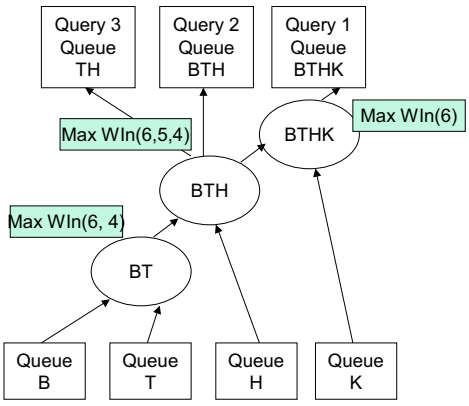

Figure 3.7: Calculating Maximum Window Size for Another Possible Query plan

- Second, it also needs a mechanism to store tuples in the operator within the window boundary.
- Finally, Juggler also needs to adapt to changing stream statistics, which is incorporated into its join algorithm.

Juggler contains structures that enable it to share computations and maintain evaluated predicate history. These structures enable Juggler to quickly process a tuple and quickly identifies if a tuple possibly satisfies multiple queries. Predicate Lists are used to group similar predicates, allowing Juggler to reduce the number of comparisons. Predicate BitSet is a structure composed of a collection of BitSets. This structure is used in several places and allows Juggler to correlate predicates to queries and tuple evaluation histories to evaluated predicates, and vice versa.

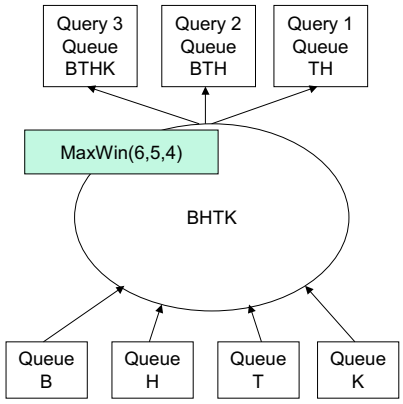

Figure 3.8: Calculating Maximum Window Size for a Query plan

```
- For each predicate type in a query {
      - For each predicate
            - Parse query into appropriate predicate structure:
              AttributePredicate or JWPredicate
      - For each Predicate List
            - If Predicate List's stream(s), column(s),
              and operator match predicate stream(s), column(s), and operator
                 - Normalize predicate
                 - If predicate is not already contained in the Predicate List
                      - Add predicate in covering order to Predicate List
                 - Set bit in QED to correlate predicate in the Predicate List
}
```

Figure 3.9: Algorithm for Registering Predicates in Juggler

3.2.1 Predicate Lists

During query registration in each operator, the predicate lists of each predicate type order similar predicates from most to least covering. Traditionally only identical predicate evaluation was shared. Similar predicate grouping allows more intermediate sharing of results between queries.

Similar attribute predicates are those that share the same stream, column, and operator. The group of similar predicates may differ in its constant and/or the need for any arithmetic operation. For example, the attributes *B.pressure = 110* and

B.pressure = 75 are similar predicates. Predicates *B.pressure =110* and *B.pressure - 50 = 45* are also similar and can be grouped after normalization. Similarly, any join predicates that share the same streams, columns, and operator can be grouped. For example, *B.pressure > T.temp* is similar to *B.pressure > 2 * T.temp* by our definition, and thus they can share tuple evaluation.

BitPos		*Stream*	*Column*	*Op*	*Value*
ListIndex	*PredIndex*				
0	0	H	beatrate	=	52
1	0	T	fluct	=	103
1	1	T	fluct	=	52.5

Figure 3.10: Attribute Predicate Lists and BitPositions

BitPos		*Stream*	*Column*	*Op*	*Stream*	*Column*
ListIndex	*PredIndex*					
0	0	B	temp	>	H	vib
1	0	B	pressure	> 2*	T	fluct
1	1	B	pressure	>	T	fluct
2	0	T	incr	=	H	vib

Figure 3.11: Join Predicate Lists and BitPositions

BitPos		*Stream*	*Column*	*Op*	*Stream*	*Column*
ListIndex	*PredIndex*					
0	0	T	ts	>	B	ts
1	0	B	ts	>	T	ts
2	0	B	ts	<	H	ts
3	0	T	ts	>	H	ts

Figure 3.12: Window Predicate Lists and BitPositions

Figures 3.10, 3.11, and 3.12 indicate the BTHK operator's predicate lists and predicates in each list. Each list groups predicates with common streams, columns, and operator. Supported operators are <, >, <=, >=, and =. Each predicate is identified by its BitPosition, coordinates that store the predicate's list index, and its position within the list. These predicate BitPositions correlate the positions of the predicates in the predicate lists with the bits in Juggler's representation of a

query object, called Query Encoding Dependency and tuple's predicate evaluation history, Join Predicate BitSet.

Predicate lists exploit similar predicates between queries. For example, Queries 1 and 2 share the predicate *B.temp* > *H.vib*. This predicate is listed only once in the join predicate list for streams: B and H; columns: temp and vib respectively; with operator: >.

During predicate registration of a new operator, described in Figure 3.9, each new predicate is first compared with the available predicate lists' streams and columns. A predicate is inserted into a predicate list when the streams, columns, and operator of both are identical. If a list is matched to the predicate, the predicate's operator is then normalized. For example, during registration of Query 1's predicate *B.pressure* > *T.fluct*, none of the predicate lists seem to match. Therefore a new list is created for the predicate's streams, columns, and operator: *B*, *T*, *pressure*, *fluct* and >. During the initialization phase, when a new predicate list is created, the predicate's left side is chosen to be the normalized side. For example, in the above mentioned predicate list, the side of *B.pressure* was chosen to be the normalized side. Later, as Query 2's predicates are registered, a predicate list match was found for the predicate *T.fluct* < *0.5 * B.pressure*. At a glance, the relevant streams and columns for the predicate list do not seem to be a match, but the operator needs to be normalized to finalize the list match. Since the list had chosen the first predicate's left side to be the normalized side, the resulting partially normalized predicate is *B.pressure * 0.5* > *T.fluct*. It is now apparent that the operator is also a match for the list.

Before the predicates are ordered in the matching list from most to least covering, they are further normalized. The structure of the predicates before normalization is of the following form: *factor1 * (str1.col1 + add1 - sub1)) operator ((str2.col2*

*+ add2 - sub2) * factor2*. Consider the join predicate list for *B pressure*, *T fluct*, and the operator >. In this list, the partially normalized predicate *B.pressure * 0.5 > T.fluct* has been normalized to *B.pressure > 2 * T.fluct*. The predicate, *T.fluct < B.pressure * 0.5* has *factor1* value of 1, *factor2* value of 0.5, and *add1*, *add2*, *sub1* and *sub2* all had a value of 0. After normalization, predicates appear as *1 * (str2.col2 + 0 - 0)) operator ((str1.col1 + addNor - subNor) * factorNor*. Continuing with our example, the normalized predicate, *B.pressure * 0.5 > T.fluct* has *factor1* value of 0.5, *factor2* value of 1, and *add1*, *add2*, *sub1* and *sub2* all had a value of 0. This allows listing the predicates in an ordered manner.

Join and window predicate lists have a side that is chosen to be the normalized side. Attribute predicates have a similar structure in comparison to the join and window predicates, but one side of the equation is a constant. Therefore, attribute predicates do not choose the normalized side. Rather, the non constant side is always normalized. For example, Query 2's attribute predicate is defined as *T.fluct * 2 = 103*. Since attribute predicates normalize the non-constant side, the resulting normalized predicate is: *T.fluct = 52.5*.

3.2.2 Juggler Predicate BitSet Structure

The Juggler Predicate BitSet Structure (JugglerPBS) is used by two other Juggler structures, Query Encoding Dependency (QED) and each tuple. It is used by each query object to correlate the predicates in the predicate lists to their queries. It is also used by each tuple to maintain a tuple's relevant predicates and its evaluation history.

Each tuple carries two Predicate BitSets, one maintains relevant predicates and the another maintains the tuple's predicate evaluation history to calculate a tuple's relevancy to the queries in the operator thus far. Most importantly, this structure

enables quick bit comparisons of the tuple's JugglerPBS and a query's QED to determine if a tuple has satisfied or could satisfy a query. These comparisons are done by using bit comparisons offered by Java's BitSet class.

The predicate bitset structure is determined by the predicate lists. Each predicate list is represented by an array of java.util.BitSets. The size of the BitSet array is determined by the number of predicates in the list. The JugglerPBS is a representation of all the predicate lists as a collection of BitSet arrays. This structure is essential for the adaptive join algorithm since it enables quick bit comparisons which identify predicates that need to evaluated. These comparisons include a combination of ANDing/ORing the corresponding BitSets in the tuple's JugglerPBS structure and the query's QED structure. To identify if a tuple has satisfied all the query's predicates before it is output, the tuple's JugglerPBS and query's QED are ANDed. If the result is identical to the query's QED the tuple has satisfied all of the query's predicates. To indicate tuple's relevant predicates to be evaluated, the query's QED and the tuple's JugglerPBS are ORed. Juggler uses these bit comparisons often during its evaluation phases.

The size of an operator's predicate bitset structure is determined after the registration of all the queries and its predicates. The config.xml file defines the operator's queries and their predicates. Each predicate is registered into the operator as they are parsed from this file. As each predicate is registered, predicate lists are created as needed.

The JugglerPBS structure is local and only relevant to the operator which contains it. When a tuple enters the operator, it is appended with one JugglerPBS. This structure represents all the predicate lists and the predicates they contain. For example, in Figure 3.12, the window predicate type has four predicate lists each containing only one predicate. This is represented by the BitSet array shown in

Figure 3.13. Query 1 contains only one window predicate, therefore only one bit is set. If there are several similar predicates in a list, there are an equal number of BitSets to represent each predicate. This is shown by the attribute's second list, which contains two BitSets, representing two predicates in this list.

The JugglerPBSs used throughout the evaluation of the tuple by the operator. It is used to indicate the predicates which the tuple has satisfied thus far. Based on this, it is also used to verify the relevancy of the queries and the remaining unevaluated predicates. Once the resulting joined tuples are sent to the output queue, the predicate bitset structure is cleared since it is no longer valid for other operators.

Query	*BTHK*	*Attr.*	*Join*	*Window*
Query1	1111	1	1	1
		00	01	0
			0	0
				0
Query2	1110	0	1	0
		01	10	1
			0	1
				0
Query3	0110	0	0	0
		10	00	0
			1	0
				1

Figure 3.13: QEDs for node BTH

3.2.3 Query Encoding Dependency Structure

A Juggler query object represents a query's predicates as BitSets. These BitSets are used to correlate the query's predicates in the predicate lists. This avoids storing predicates in multiple places. The query object contains a JugglerPBS and names of the streams it pertains to. The query object is referred to as the Query Encoding Dependency (QED). Thus, Juggler's query representation is a collection of java.util.BitSets, where each BitSet corresponds to a BitPosition. A BitPosition is

a coordinate composed of the predicate list index and the index of the predicate within the list. Therefore, the structure of a query's QED is determined by the number of predicate lists and the number of predicates in each list. If a predicate is relevant to a query, the BitSet in the corresponding location of the predicate's BitPosition is set to 1 to indicate its relevancy. A BitPosition set to 0 indicates the predicate in the corresponding location is not relevant to the query. The tuple's JugglerPBS and query's QED is used to efficiently evaluate relevant predicates and maintain the tuple's queries that it may satisfy. The evaluation consists of bit comparisons between these structures to identify tuples that have satisfied a query and also identify predicates that are to be evaluated on a tuple.

In order to output only the tuples that satisfied a query, the QED is needed to define the attribute, window, and join predicates' *relevancy* to a query. This also allows for optimization. If evaluating the predicate types in the Attribute-Window-Join order and a tuple satisfies the attribute predicates listed in a query's QED, the irrelevant join and window predicate bits are cleared to 0. These predicate bits are identified by using the query's QED. Each bit set in the QED is cleared in the tuple's RelPBS, the bitset that represents tuple's relevant predicates to be evaluated. This is done using a combination of bit manipulations, ANDing, ORing, and XORing. Comparisons of unnecessary predicates are thus avoided. The Query Encoding Dependency also helps to eliminate outputting tuples that may be false positives since a tuple is only output if its predicate bit encodings satisfy at least one query. If the tuple satisfies several queries, it is only output once. The tuple will carry query IDs for all the queries it has satisfied.

The QED of a composite operator such as BTHK describes the relationship of the predicates and the queries as shown in Figure 3.13. Figures 3.10, 3.11, and 3.12 indicate the predicates represented by each bit in the BTHK's QED. BitPosition

i:j for attribute predicate type corresponds to the j^{th} predicate in the i^{th} attribute predicate list, as shown in Figure 3.10. The join and window predicates are also similarly encoded as illustrated in Figures 3.11 and 3.12. Each bit in the QED corresponds to only one predicate in either the attribute, join, or window bit encodings. The predicates are attached as a collection of BitSet arrays to each tuple.

3.2.4 Predicate BitSet Structure

Attr.	*Join*	*Window*
0	0	0
00	00	0
	0	0
		0

Figure 3.14: Juggler Tuple's Relevant Predicate BitSet Structure (RelPBS)

Every tuple contains two sets of Predicate BitSet structures. This is to maintain the satisfied and relevant predicates. Before every predicate type evaluation and before the tuple is output, the tuple's JugglerPBS is compared to its relevant queries' QEDs. This determines if the tuple has satisfied the queries' predicate requirements.

The two sets of the predicate BitSet structures are relevant predicate bitset (RelPBS) and satisfied predicate bitset (SatPBS). RelPBS for the node BTHK is shown in Figure 3.14. SatPBS has the same structure as RelPBS, but RelPBS maintains the relevant predicates to be evaluated and SatPBS maintains the predicates satisfied by the tuple thus far at each predicate evaluation phase. When a BitSet is set to 1 in a tuple's RelPBS, this designates the predicate in the corresponding BitPosition is relevant and needs to be evaluated. When a BitSet is set to 0 in a tuple's RelPBS, this indicates the predicate in the corresponding BitPosition is not relevant and need not be evaluated. The SatPBS is only relevant if the bits in the corresponding RelPBS are set. In this case, if a bit in the tuple's SatPBS is set to 1,

this indicates the tuple has satisfied the predicate in the corresponding BitPosition. When it is set to 0, this designates the tuple has not satisfied the predicate. Note, both RelPBS and SatPBS have identical structures since they both represent the operator's predicate lists and the predicates in each list.

```
Each predicate list is represented by a BitSet,
which is an array of bits.
Comparisons: AND, OR, XOR, are BitSet comparisons.

\\To identify if a tuple has satisfied a query
- result = query's QED AND tuple's RelPBS
- If result == query's QED
     - result2 = query's QED and tuple's SatPBS
     - If result2 == query's QED
       - Output tuple for query

\\To identify a candidate query for a tuple
  \\Only compare tuples that relevant
  - result = query's QED AND tuple's RelPBS
  - result2 = result AND tuple's SatPBS
  - If result 2 == result
    - Tuple has satisfied query predicates
      that were relevant to the tuple
           query is a candidate query
  - Else
     query is not a candidate query
     tuple has not satisfied a relevant and
     required predicate

\\To identify tuple's irrelevant predicates
- result = query's QED AND tuple's RelPBS
- result2 = result XOR result
//To maintain the predicates that have been evaluated
- result3 = result2 OR tuple's SatPBS
- tuple's RelPBS = result3

\\To identify tuple's relevant predicates
- result = query's QED OR tuple's RelPBS
- Tuple's RelPBS = result
```

Figure 3.15: Bit Comparisons with QEDs and tuple's JugglerPBS

Initially, when a tuple enters the operator, all RelPBSs are set to indicate that all predicates of the first most selective predicate type are relevant and need to be evaluated. During the predicate phase evaluation, the satisfied predicates for the predicate type are set in the SatPBS whenever a predicate evaluates to true. After the predicate evaluation phase is completed, the tuple's SatPBS resulting from the current phase will be compared to the relevant queries' QED. This will indicate

which predicates the tuple has satisfied and which queries it may possibly satisfy. The tuple's candidate query IDs are updated, and these queries' QEDs are used to set the tuple's RelPBS for the next predicate type. Each candidate query's QED for the next predicate type is combined with the tuple's RelPBS for the corresponding predicate type by ORing BitSets. the tuple's RelPBS will indicate the relevant predicates to be evaluated during the next predicate evaluation phase reducing the number of unnecessary computations.

For example, if a tuple has satisfied window predicates in the BitPositions 0:0 and 1:0, the only candidate query found by comparing the tuple's SatPBS and the queries' QED is Query 3, Figure 3.15. Although the tuple has satisfied the predicate in BitPosition 1:0, it has not satisfied the predicate in BitPosition 2:0. This eliminates Query 2 as a candidate query. The updated candidate queries are used to set the next phase's relevant predicates. In this case, only predicates relevant to Query 3 will be set and evaluated in the next evaluation phase, thus reducing the number of comparisons.

		Stream H		*Stream K*			*Stream B*
ts	*beatrate*	*vib*	*ts*	*fluid*	*ts*	*press*	*temp*
0	6	98	1	150	1	150	101
1	1	90	2	100	2	97	101
2	7	98	3	150	4	35	101
3	2	98	5	150	5	100	101
4	8	89	6	75	8	150	101

Figure 3.16: Tuples In Juggler Operator BTHK

3.3 Storing Tuples in Operator

Juggler must store tuples within the current window to join them with newly incoming tuples typically referred to as "state". It also needs a mechanism to quickly retrieve tuples to join. Tuples in Juggler are stored in two places: Join Exploita-

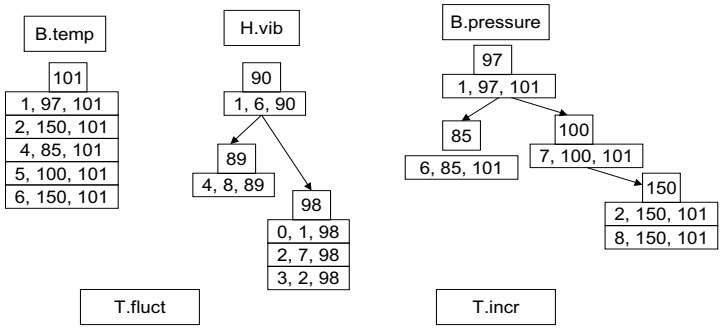

Figure 3.17: Join Exploitation Structures (JESs) in BTHK

tion Structure and Window Exploitation Structure. Join Exploitation Structures are used when join predicates are being evaluated, namely to retrieve tuples to join with. Similarly, the Window Exploitation Structures are used to retrieve tuples to join with when window predicates are being evaluated. These structures store tuples within the current window, i.e. the current state. They also allow quick retrieval of tuples based on a value for value based join processing or for time-based join predicates.

3.3.1 Join Exploitation Structures

To allow for efficient computation of theta joins, one structure, Join Exploitation Structure (JES), a collection of Red-Black trees, is needed in conjunction with the predicate lists. This is a list of covering predicates for a stream and its column that will indicate if multiple predicates are concurrently satisfied by a tuple. In our running example, the join predicate represented by the second list's first position, BitPosition 1:0, is a theta join, Figure 3.11. It is a binary expression comparing a

stream's column to an expression involving another stream's column: *B.pressure > 2 * T.fluct*. Normalized predicates allow for theta joins with additions, subtractions, fractions, and the combinations of all three.

In our example, in Figure 3.6, the operator BTHK contains four join exploitation structures. Each exploitation structure represents a stream and column pair in registered predicates in the operator. In other words, for each stream and column involved in a join predicate, there is a corresponding JES. One such exploitation structure is for the data stream blood (B) and its pressure column, (rightmost tree in Figure 3.17). Maximum number of possible JESs in an operator is the number of predicates * 2. In this case, none of the predicates have any identical stream and column pairs. If there are stream-column pairs that are in more than one predicate, the number of JESs in an operator is reduced. Red-Black trees, like those shown in Figure 3.17, will be used to sort the values of incoming tuples so that those within a specified range can be quickly retrieved. For example, when processing the join between *B.pressure* and *T.fluct*, tuple T.fluct's value can be used to retrieve candidate B's tuples whose pressure values are greater than half of its value. This only allows for efficient retrieval of candidate tuples that will satisfy the predicate. It thus also reduces the number of intermediate joined tuples compared to the number of intermediate tuples generated if all the streams tuples in the operator were retrieved and joined.

3.3.2 Window Exploitation Structure

Similar to join predicates, another data structure in conjunction with window predicate lists is needed to allow for both purging based on window range as well as filtering and joining based on theta window predicates. These are join predicates that are joined based on tuple timestamps. A list of window predicates in order of

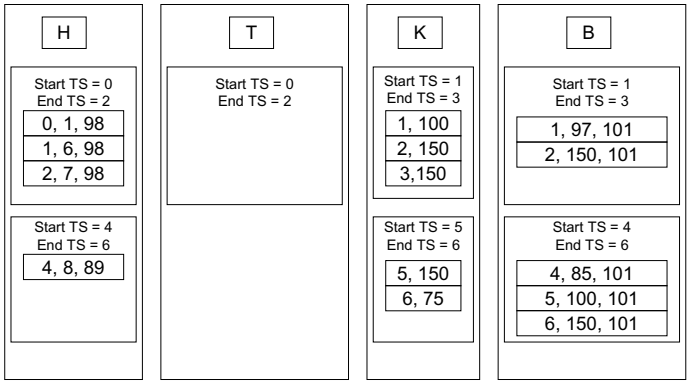

Figure 3.18: Window Exploitation Structures for Juggler Operator BTHK

most to least covering for a stream's column is used to reduce the number of comparisons. The predicate list for B, *pressure*, T and *fluct* in Figure 3.11, is an example of a list that contains predicates in order of most to least covering. In Juggler's current implementation, a predicate with a higher factor is considered more covering. If the factors are equivalent, then the addends are compared. The predicate with the higher addend is considered more covering. This is repeated for the minuends. Juggler has only tested predicates with a factor or addend. Juggler's tests have not violated the ordering algorithm since the data distributions were known and predicates were created with this information in consideration. This restriction can be relaxed in the future.

Tuples are assumed to arrive in order of their timestamps. A Window Exploitation Structure (WES) groups tuples into a buffer to efficiently access relevant tuples. Each WES can contain a maximum number of buckets defined by the user during operator's initialization. If a user desires each bucket to represent one time unit, the user must define the number of buckets equal to the operator's window size. For

example, if an operator has a general window size of 6 time units and each window bucket contains tuples within a range of 3 time units, only two buckets per WES are needed. The bucket structures not only allow quick removal of stale tuples but also provide a mechanism to retrieve tuples to join with for a given timestamp.

To join tuples using window predicates, one stream's timestamp value can be used to access the other stream's values by retrieving tuples from its appropriate buckets. Note, Juggler uses tuple's maximum timestamps to compare values. The maximum window size for each query is defined in a SQL-like query description, Figure 3.3. The maximum of the queries' window sizes is used to determine the operator's maximum window size, as shown in Figures 3.7 and 3.8. This size determines the range of time units that a WES will store. The number of buckets is pre-defined for the operator, allowing the user to define number of buckets in each operator. Given queries' window predicates, there is an opportunity for future work to change bucket size dynamically to reflect the changing stream arrival rates. Since buckets are only an efficient mechanism to retrieve tuples within a range, the ability to change WESs bucket granularity could further optimize this structure.

The maximum window range is divided equally into the number of buckets. Each bucket maintains the start and the end timestamp of the tuples it contains. The range of the tuples contained in a bucket is less than or equal to the operator's maximum window size divided by the number of buckets. The tuples in the WESs are sorted by timestamp, guaranteeing the tuples' timestamp values are within the WES's range.

Assuming the tuple has not yet been joined, when the appropriate WES is found for the predicate, each bucket's start and end timestamp is used to compare if the bucket could be relevant to the predicate and the input tuple's timestamp. If the bucket's start and end timestamps satisfy the predicate, all the tuples in the bucket

are retrieved to join with the input tuple. After that, the algorithm proceeds to the next bucket for tuple retrieval. If the start timestamp of the bucket satisfies the predicate, but the end timestamp does not, then only the relevant tuples in the bucket are retrieved. If both the start and end timestamps do not satisfy the predicate, none of the tuples in the bucket are retrieved.

A WES also provides a quick way for removing stale tuples. When the WESs time range exceeds maximum window size, the tuples in the first partition are removed and a new partition is created. This algorithm assumes that the tuples arrive in order of their timestamp. Input tuples that are already a join tuple, an output of another Juggler operator, is ordered by its maximum timestamp. Handling cases when tuples arrive out of order will be left as future work.

In our example, the node BTHK's maximum window size is six seconds. If the incoming data tuples are partitioned into two buckets, the most recent incoming tuples are contained in the last bucket of the WESs. Using these buckets, the number of candidate tuples within a specified window range are quickly retrieved. For example, for Query 2, one of the window predicates specifies $B.ts$ must be less than $T.ts$. If T's ts value is located in the first bucket, then B candidate tuples will only be retrieved from the first bucket and the other bucket can be ignored.

3.4 Adaptive Predicate Ordering

3.4.1 Tuple's JugglerPBS

There are three types of predicates a query may specify: attribute, window and join. A tuple entering an operator must undergo all three predicate type evaluations assigned for this operator before it can be output. Predicate types to be applied to a tuple are ordered by their selectivity as the stream value distributions change.

To maintain which predicates have already been evaluated and which out of those a tuple has satisfied, a JugglerPBS is attached to each tuple. JugglerPBS is composed of two identical structures. One structure, Relevant Predicate BitSet (RelPBS), maintains tuple's relevant predicates. The other structure, Join Predicate BitSet (JugglerPBS), maintains tuple's evaluated and satisfied predicates. Each of these structures is an array of BitSets representing the predicate lists and each bit represents predicates in the list.

Note that since an operator can be shared, it may have multiple parents. In Figure 3.5, the operator BTH is shared by two queries: Query 1 and 2. Query 1 takes relevant tuples that are output by the operator BTH and joins these tuples with those from the kidney (K) data stream. JugglerPBSs are used to associate tuples with the satisfied queries they have satisfied and vice versa. This allows an operator to process multiple queries and output relevant tuples for each accordingly.

3.4.2 Predicate Ordering

The order of applying the predicate types can be interchanged. Six orderings are possible: Attribute-Join-Window (AJW), Attribute-Window-Join (AWJ), Join-Attribute-Window (JAW), Join-Window-Attribute (JWA), Window-Attribute-Join (WAJ), and Window-Join-Attribute (WJA). The first predicate type applied maximizes the efficiency of its exploitation structures and is selected for being the most selective of the three, reducing the list of predicate evaluations for the intermediate tuples. Exploitation structures are only used by its corresponding predicate type. The second and third predicate type will only evaluate predicates indicated as relevant on the intermediate tuples found by the previous predicate type.

The predicates applied to these tuples after the first evaluation stage will only be relevant to the tuple's candidate queries. These are queries that a tuple may

possibly satisfy. If the first predicate type can reduce the number of candidate queries, the number of subsequent predicate type evaluations will also be reduced. The resulting output will be the same regardless of the predicate type orderings, but applying the predicate types in order of selectivity can greatly reduce the number of computations and intermediate tuples.

Figure 3.19 outlines the algorithm for ordering of the three predicate types. All six orderings of the three predicates are incorporated into the algorithm.

```
For each predicate type {
    - If first and most selective predicate type {
        - Set all tuple's Relevant Predicate BitSets (RelPBSs) to true
    }
    - Evaluate predicate type
        - For all relevant predicates indicated by tuple's RelPBS{
            - If predicate type is not attribute {
              - Retrieve candidate tuples to join using the join/window exploitation structures
            }
            - Binary Search on the predicate list containing the relevant predicate to find the most covering
            - Set bits in tuple's Satisfied Predicate BitSet (SatPBS) for all covering bits for the predicate list
        }
    - Compare with Query Encoding Dependency (QED)
        - If tuple's JugglerPBS encoding is not a superset of any query QED, delete tuple
        - If superset, set tuple's RelPBS of the remaining predicate types
          using candidate query's QED
    - Store candidate tuples in intermediate buffer, storing joined tuples during processing
}

Join remaining candidate tuples:
    - Combine tuple's and candidate tuple's attribute JugglerPBS by ORing BitSets
    - Apply all remaining attribute predicates that have not been evaluated,
      identified by tuple's RelPBS and JugglerPBS
        (This is the case when a stream was not involved in a predicate.
        All the tuples of that stream become candidate tuples.
        These tuples may not have had some predicates evaluated.)

Output joined tuples to output buffer(s) for appropriate queries the tuple has satisfied

Insert input tuples into Exploitation Structures

Clean-up Exploitation Structures to remove stale tuples
```

Figure 3.19: Join Algorithm

```
-For each query in the tuple's query list{
     - Compare tuple's streams with query's streams
     - If tuple does not contain all of query's streams
          - Retrieve all the tuples of the missing stream
          - Join tuple with the tuples of the missing stream

     - If tuple has not evaluated all of query's predicates
     //compare tuple's RelPBS and query's QED
          - Evaluate these predicates
     - If tuple has satisfied all query predicates
     //if tuple's SatPBS is a superset of query's QED
          - Output tuple for this query
     - Else
          - Remove query from tuple's query list
          - If tuple's query list is empty
               - Remove tuple from output tuple list
}

\\Route tuples
\\During initialization each output queue is associated with
\\ query/queries output buffer
\\ Correlation of query and its index in operator's  array of output queues
\\ is stored a hash table

For each query remaining in tuple's query list{
    -  Find output queue index for query using hash table
    - Delete tuple's JugglerPBS structure \\ this is a local structure, only relevant to the operator
    -  Add copy of tuple to output queue
}
```

Figure 3.20: Tuple Output Algorithm

3.4.3 Output Tuples

A tuple is output if it has satisfied at least one query. Before a tuple is output, each of the queries in the tuple's list is used to verify that the tuple contains all the streams defined in the query's *From* clause and that all the query's predicates have been evaluated and satisfied. For example, if an operator includes a query which contains the streams, A, B and C. Assume the operator also includes a second query which contains streams A and B. Tuples, AB, that have satisfied the second query cannot be output for the first query since stream C is missing. If a tuple AB satisfies both queries, and the first query does not include a join predicate where tuples of stream C were not retrieved, this tuple needs to join with the missing stream. If the tuple is missing a stream, the missing stream's tuples within the time window are

retrieved and joined with this tuple.

Before the joined tuples are output, tuple's JugglerPBS and queries' QED are compared to ensure all predicates have been evaluated. Any predicates that may not have been evaluated since the tuple was found to be missing a stream, will be evaluated at this time. After the evaluation, the tuple's evaluated and satisfied predicates are compared to the query's predicates. If all the predicates have been satisfied, the tuple is output for this query. If not, the query is removed from the tuple's query list. This algorithm is described in more detail in Figure 3.20.

3.4.4 Insert Tuples into Exploitation Structures

```
-For each tuple{
   //Enter tuple in WES
     //Find appropriate WES
     - For each WES{
       - If WES's stream(s) == tuple's stream(s)
          - Retrieve last bucket
          - Enter copy of tuple in order of timestamp
          - break;
    }
   //Enter tuple in JES
     //Find appropriate JESs
     - For each JES {
       - If JES's stream == tuple's stream
         - valueCol = Retrieve tuple's value for JES's column
         - Enter copy of tuple in JES, with value of valueCol
     }
}
```

Figure 3.21: Inserting Tuples into Exploitation Structures Algorithm

After joined tuples have been output, the tuples that had entered the operator must be entered into the exploitation structures to be joined with future input tuples. This algorithm is outlined in Figure 3.21. The tuple is entered in every JES that includes the tuple's stream. The tuple is entered in the JES on the its value of the JES's column.

All tuples are also entered into a WES. Each WES is traversed until one is found that includes the tuple's stream(s). Since, Juggler assumes that tuples arrive in

order of timestamp, the tuple will be entered into the last bucket of the WES. Note, copies of tuples are entered into the JESs and WES. To optimize these structures in the future, references to the tuple should be stored.

3.4.5 Clean Up

```
-For each input tuple{
     - For each tuple stream's WES bucket
          - If tuple falls within the bucket time range
             - Store tuple in this bucket
             - Return, no clean up is needed
     - If a bucket for the tuple has not been found
          - Create a new bucket and increment the bucket count
          - If WES's time range has exceeded operator's maximum window,
             indicating stale tuples
             - The first stream bucket is purged
             - For each tuple in the purged bucket
                  - For each JES
                        - If the JES's stream and tuple's stream are a match
                             - Remove copy of tuple from JES
}
```

Figure 3.22: Juggler Clean Up Algorithm

After tuples are output, the operator maintains its window size. The input tuples must be entered into the operator to join with incoming data. There are two conditions that activate the clean up process. The first condition occurs when the range of the minimum and maximum timestamps of the WESs is greater than the operator's maximum window. In this case, the buckets in the WESs are purged until the range falls within the operator's window size. Buckets are only used to partition the tuples within the operator's window size. The second condition occurs when an input tuple requires the creation of a new WES bucket, incrementing the stream's WES bucket count. If the WES's time range exceeds the operator's maximum window size, the stream's first bucket is purged. This maintains the window range of the operator.

When a WES bucket is purged, the tuples must also be removed from the JESs. Each tuple being purged is removed from all the JESs that contain the tuple's

stream. A copy of the tuple in the JES is identified by its values and timestamp. This avoids removing tuples that may have identical values from the JESs. This clean up process is described in Figure 3.22. The JESs are a collection of Red-Black trees in which current Juggler implementation stores copies of tuples in the JESs. This is a drawback of using Java's Red-Black trees. In the future, another data structure or tuple referencing scheme should be used to reduce this overhead.

3.4.6 Juggler Architecture Overview

Juggler is composed of several data structures used in Juggler's adaptive predicate ordering algorithm. These structures enable Juggler to reduce the number of computations when processing a tuple and share sub computations between queries. Computation is also shared by Juggler's multiple functionalities, specifically due to the combination of joins and selects into one operator.

One of Juggler's data structures is a Predicate List. Predicate Lists are contained in each of the predicate types. Predicate Lists allow the operator to group similar predicates and order them from most to least covering. This enables the operator to evaluate multiple predicates at a time and quickly narrow down evaluation to only the relevant predicates. As the most covering predicate is found, tuple's SatPBS bits are set to reflect all the predicates the tuple has concurrently satisfied.

Exploitation structures are also an integral part of Juggler. Only the predicate types that result in joined tuples have associated exploitation structures. These consist of join and window predicate types. During each evaluation phase, Predicate Lists and the QED bits are compared to assess the tuple's candidacy for the queries. The interaction of these structures are key to Juggler's adaptive join algorithm.

Juggler's components and its logical architecture is shown in Figure 3.23. The operator's intermediate buffer, an array, stores the joined tuples during the tuple

processing is not shown in Figure 3.23. All the structures interact with each other during Juggler's join algorithm, especially during join processing, tuple filtering, and predicate evaluation. The details are illustrated in the running example.

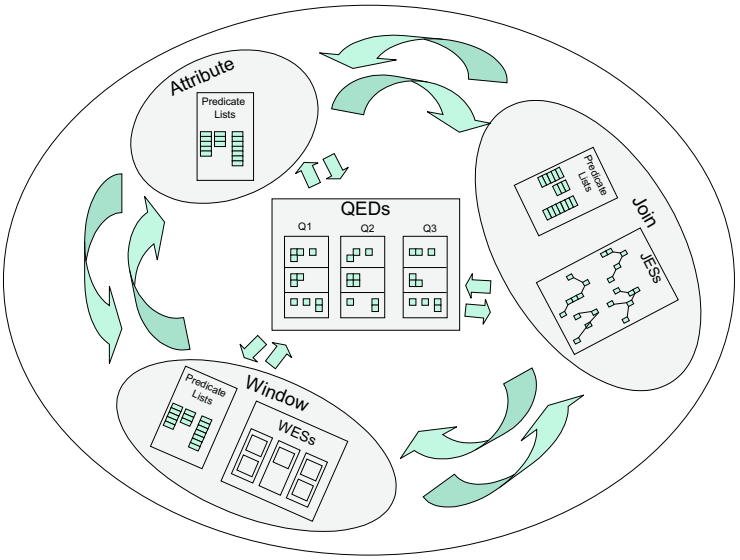

Figure 3.23: Logical Architecture of Juggler Operator

Chapter 4

Running Example

In our running example, the predicate types will be applied in this order of selectivity: window, attribute, and join (WAJ). Selectivity for each predicate type is calculated while tuples are evaluated in the operator. The number of tuples given to a predicate manager and the number of tuples resulting determine the predicate type's current selectivity.

The running example will follow the algorithm outlined in Figure 3.19, and each step will be described in greater detail. To begin the algorithm, we will assume that the tuple arrives in operator BTHK's input queue, T. The new incoming tuple has values of (1, 103, 101, 98) for the columns *ts, fluct, degree, incr*, respectively. CAPE's execution controller will detect a tuple has arrived on an input queue and call BTHK's run method. This method will retrieve the tuples from the input queues.

Assuming tuples in Figure 3.16 from streams B, H, and K have been previously evaluated by the operator, the resulting JESs and WESs are populated as shown in Figures 3.17 and 3.18. We will also assume that there has not been another tuple T that had previously been evaluated by the operator. Hence, JESs and WESs for

stream T are empty. This is to simplify the description of the clean up process and the storage of the tuples in the exploitation structures.

4.1 Optimal Selective Predicate Ordering

4.1.1 First Predicate Type Evaluation

	RelPBS				SatPBS	
Attr.	Relevant *Join*	*Window*		*Attr.*	Satisfied *Join*	*Window*
0	0	0		0	0	0
00	00	0		00	00	0
	0	0			0	0
		0				0

Figure 4.1: Initial Juggler Predicate BitSet Structure (JugglerPBS)

	RelPBS				SatPBS	
Attr.	Relevant *Join*	*Window*		*Attr.*	Satisfied *Join*	*Window*
0	0	1		0	0	0
00	00	1		00	00	0
	0	1			0	0
		1				0

Figure 4.2: JugglerPBSbefore First Predicate Type Evaluation in the WAJ Predicate Ordering

	Stream B	
ts	*press*	*temp*
2	150	101
6	85	101
7	100	101
8	150	101

Figure 4.3: First Window Bucket of Stream B

We will assume that the most selective predicate type is the window. Before an incoming tuple is processed, a new JugglerPBSas in Figure 4.1 is created each time and associated with the tuple. All the bits are initially set to 0 to indicate

that no predicates have been evaluated or satisfied. The bits for the first evaluated predicate type of RelPBS, which is window in our example, are all set to true, as shown in Figure 4.2. For the first relevant predicate type evaluation, all predicates need to be evaluated to be able to assess potential queries the tuple may satisfy. The first predicate evaluation phase will retrieve a set of intermediate joined tuples from its corresponding exploitation structure and also determine candidate queries for each. The predicate list for this predicate type will be traversed to evaluate predicates and retrieve candidate tuples to join from the corresponding exploitation structures. The predicate in the first window predicate list is $T.ts > B.ts$. In our simple running example, our tuple T has a timestamp value of 1. Once we have extracted the tuple's timestamp value, which is 1, the predicate then becomes $B.ts < 1$. This value is used to retrieve the satisfying tuples from B's WES. The start timestamp of B's first bucket quickly indicates that there are no relevant tuples that can be retrieved for this predicate. The next predicate list evaluated is $B.ts > T.ts$. Using the newly incoming tuple T's value, the predicate becomes $B.ts > 1$. Using B's WES, the start and end timestamps of both buckets, 1, 4 and 3, 6 respectively, indicate this time that there may be some relevant tuples in the first bucket and that all tuples in the second bucket satisfy the predicate. After traversing the first bucket and retrieving all the tuples in the second, the resulting tuples to join are listed in Figure 4.3.

The algorithm continues to traverse the window predicate lists, Figure 3.12. The next list contains the predicate: $B.ts < H.ts$. Since the tuple being processed is T, the tuple's stream is not included in the predicate streams, B and H. This predicate is unable to be evaluated. The RelPBS in the Bit Position 2:0 is unset to indicate the predicate is not relevant to this tuple. The last list contains the predicate: $T.ts > H.ts$. Using the incoming tuple T's value of 1, the predicate becomes $H.ts < 1$. The

WES of H in Figure 3.18 has a start timestamp of 0 and an end timestamp of 2 for its first bucket which implies only the first bucket contains possibly relevant H tuples. Only one tuple is retrieved, H: (0, 1, 98). Tuples added to the candidate tuple list, which will be joined to tuple T resulting in Figure 4.4. During processing, these tuples are joined on the fly because the retrieved tuple must be able to maintain its JugglerPBS for future joins with other stream tuples. Before a tuple is output, the tuples are joined. Before this tuple is added to the candidate tuple list, the list is checked to guarantee that this tuple has not already been added during a previous predicate evaluation. Each tuple in the list is traversed and the tuples' timestamps, streams, and values are compared to eliminate the possibility of duplicates. Since the current candidate list only contains tuples from stream B, tuple H can be added without the possibility of duplication.

After the predicate lists have been traversed, all predicates for each of the five candidate tuples are re-evaluated. This is to evaluate predicates that were not evaluated in the previous step. When there are more queries with overlapping predicates, each tuple needs to have all relevant predicates evaluated in order to calculate the tuple's potentially satisfied queries. All relevant predicates are indicated by the tuple's set bits in its RelPBS.

Figure 4.4 lists all the intermediate joined tuples, which are set of stream sub-tuples, that resulted during the first predicate type evaluation. It also lists the JugglerPBS for each tuple indicating the relevant and satisfied window predicates. Since this is the first predicate type to be evaluated, tuple's relevancy to all the registered queries in the operator will be evaluated to populate the query list with candidate query IDs.

To update the query list for each tuple after the predicate type evaluation, the tuple's JugglerPBS and each registered query's QED are compared. The first step

Str									*Attr*	Relevant *Join*	*Win*		*Attr*	Satisfied *Join*	*Win*
TB	*ts*	*fluct*	*deg*	*incr*	*ts*	*press*	*temp*			RelPBS				SatPBS	
TB	1	103	101	98	2	150	101		0	0	1		0	0	0
									00	00	0		00	00	0
										0	0			0	0
											1				1
TB	1	103	101	98	4	85	101		0	0	1		0	0	0
									00	00	0		00	00	0
										0	0			0	0
											1				1
TB	1	103	101	98	5	100	101		0	0	1		0	0	0
									00	00	0		00	00	0
										0	0			0	0
											1				1
TB	1	103	101	98	6	150	101		0	0	1		0	0	0
									00	00	0		00	00	0
										0	0			0	0
											1				1
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*		RelPBS				SatPBS		
TH	1	103	101	98	0	1	98		0	0	0		0	0	0
									00	00	1		00	00	1
										0	0			0	0
											0				0

Figure 4.4: Intermediate Tuples After First Predicate Type Evaluation in WAJ Predicate Ordering

in this evaluation process compares a QED with the tuple's RelPBS. If the BitSets in both the QED and tuple's RelPBS are set, indicating the predicates in the corresponding positions were not relevant and therefore not evaluated, these predicates are ignored in this evaluation. If there are BitSets in which both the QED and tuple's RelPBS are set, indicating the predicate is relevant to the tuple and was evaluated, then the QED and the tuple's SatPBS are compared. In this case, only the BitSets set in both the QED and tuple's RelPBS bits are considered for the tuple's SatPBS comparison, which is referred to as tempBS. TempBS will be a subset of the BitSets in the query's QED. If tempBS's BitSets are also set in the tuple's SatPBS, indicating the tuple satisfied all the predicates that were relevant and required by the query, the query remains in the tuple's candidate query list. If not, the tuple has not satisfied one or more predicates that were relevant and required by the query and its query ID is removed from the tuple's query list. Considering

only the BitSets relevant to the query, if the RelPBS and SatPBS are identical and subsets of the query's QED, the query is still a candidate query. This tuple may satisfy the remaining required predicates for the query in later predicate evaluation phases.

For example, the intermediate tuple TB (1, 103, 101, 98, 2, 150, 101) has the BitPositions of 0:0 and 1:0 of the window RelPBS set. Query 1's QED has the BitPosition of 0:0 also set. This indicates that Query 1 requires the corresponding predicate in the window predicate list to be satisfied. The BitPosition 0:0 represents the predicate $T.ts > B.ts$ indicating the BitSet in the tuple's SatPBS in the corresponding BitPosition is not set, the tuple has not satisfied this predicate; tuple has failed this predicate. Query 1's ID is removed from the tuple TB's candidate query list.

Query 2's QED and tuple TB's RelPBS do not have any matching BitSets set. Therefore either this tuple has not yet violated any required query predicates, or all of the predicates for this query may not have been relevant to this tuple. Query 2's ID will remain in the tuple's list.

Query 3 and TB both have the BitSets in BitPosition 1:0 set. The corresponding predicate is $B.ts > T.ts$. BitPosition 1:0 is also set in the tuple's SatPBS. Therefore Query 3's ID remains in the tuple's list. The other three TB tuples undergo an analogous evaluation process resulting in Queries 2 and 3 as candidate queries after the first evaluation phase.

Tuple TH will be similarly evaluated. Query 1's QED does not match any of the set BitSets in the tuple's RelPBS. This indicates that the tuples required by Query 1 were not relevant since the tuple streams did not match the predicates' streams. These predicates could possibly be evaluated at a later time when the tuple is joined with the appropriate stream. Therefore, Query 1's ID remains in the tuple's list.

Query 2's evaluation is similar to Query 1 for this tuple. Therefore, Query 2's ID also remains in this list. Query 3's QED contains a match for BitPosition 3:0, which has a corresponding predicate, $T.ts > H.ts$. The tuple's SatPBS indicates that this predicate is satisfied. Therefore, Queries 1, 2, and 3 all remain in the tuple TH's query list.

Before the next predicate type is processed and after all the tuples have had their query lists updated, the tuple's streams are compared to the queries' streams. If the intermediate joined tuple contains a stream not included in a query, the query is removed from its candidate query list. For example, tuples TB contain Queries 2 and 3's IDs in their candidate query list. The *From* clause of Query 3, as defined in Figure 3.3, indicates that the query only involves streams T and H. Stream B is not included in this query. For this reason, Query 3's ID is removed from the tuple's query list. A summary of the tuple's query list evaluation is given in Figure 4.5.

All relevant predicates need to be evaluated for each of the candidate queries. The algorithm is outlined in Figure 3.15.

									Query List
TB	*ts*	*fluct*	*deg*	*incr*	*ts*	*press*	*temp*		
TB	1	103	101	98	2	150	101		2
TB	1	103	101	98	4	85	101		2
TB	1	103	101	98	5	100	101		2
TB	1	103	101	98	6	150	101		2
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*		
TH	1	103	101	98	0	1	98		1, 2, 3

Figure 4.5: Intermediate Joined Tuples' Query List After First Predicate Type Evaluation in WAJ Predicate Ordering

4.2 Second Predicate Type Evaluation

The next predicate type to be evaluated for the predicate ordering of window-attribute-join is attribute (WAJ). This predicate type's evaluation process slightly

differs from the evaluation of join or window predicate types. Regardless of its predicate type ordering, attribute predicate evaluation does not result in more intermediate joined tuples. Also, this predicate type does not have exploitation structures. Evaluating attribute predicates requires a simple comparison of the tuple's value with the predicate's constant.

Since the four TB tuples, listed in Figure 4.5, only have Query 2's ID in the tuple's query list, Query 2's QED is used to set the appropriate predicate type of BitSets tuple's RelPBS. Query 2's QED contains only two BitSets in the BitPositions 0:0 and 1:0, as shown in Figure 3.13. Therefore these two BitSets in the BitPositions are set in the tuple's RelPBS.

Similarly, tuple TH's appropriate attribute predicate type of RelPBS is also set using Queries 1, 2, and 3. Figure 4.6 summarizes the resulting RelPBS for each intermediate tuple.

										Relevant					Satisfied	
									Attr	*Join*	*Win*		*Attr*	*Join*	*Win*
TB	*ts*	*fluct*	*deg*	*incr*	*ts*	*press*	*temp*			RelPBS				SatPBS	
TB	1	103	101	98	2	150	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	4	85	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	5	100	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	6	150	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*			RelPBS				SatPBS	
TH	1	103	101	98	0	1	98		1	0	0		0	0	0
									11	00	1		00	00	1
										0	0			0	0
											1				1

Figure 4.6: Tuples' RelPBS for Second Predicate Type in WAJ Predicate Ordering

A similar process to the one described in the previous window predicate phase

is conducted to evaluate relevant predicates with the joined tuples. Tuple TB's only set RelPBS attribute bit is in BitPosition 1:0. The corresponding predicate is: *T.fluct = 52.5*, the only relevant predicate for this tuple. This predicate is not satisfied since tuple TB's fluct value is 103. The BitSet of SatPBS in the predicate's BitPosition 1:0 is not set, indicating the predicate evaluation failed. The remaining three TB tuples are evaluated in a similar manner.

```
\\set initial predicate index in predicate list
- index = predicate list size / 2
- current predicate list size = predicate list /2
- While (true){
  - Evaluate predicate
    - If predicate is satisfied
      - If predicate includes the '=' operator
        - set appropriate bit in tuple's SatPBS
      - Else \\if operator is not '='
        \\ predicate is satisfied, there may be a more covering predicate
        - index = current predicate list size + (current predicate list size / 2)
    - Else \\if predicate is not satisfied
      \\ there may be a less covering predicate
       - index = current predicate list size - (current predicate list size / 2)
  - If current predicate list size == 0
    break; \\ reached end of list
}
```

Figure 4.7: Find Most Covering Predicate Algorithm

The first relevant predicate evaluated for tuple TH is: *H.beatrate = 52*. This predicate is satisfied. Therefore the SatPBS in the predicate's BitPosition of 0:0 is set. If there are several predicates that are relevant from a list, these predicates will be evaluated in a binary search using the findMostCovering method, outlined in Figure 4.7. To evaluate similar predicates the predicate list, the initial current predicate index is set to the size of the predicate list. Once the binary search to find the most covering predicate has begun, the current predicate index will be determined by dividing the current index by 2. Therefore, the first predicate evaluated will be at the index: *predicate list size / 2*. In our example, predicate list size is 2.

To find the most covering predicate in the predicate list of B, *pressure*, T and

										Relevant				Satisfied	
									Attr	*Join*	*Win*		*Attr*	*Join*	*Win*
TB	*ts*	*fluct*	*deg*	*incr*	*ts*	*press*	*temp*			RelPBS				SatPBS	
TB	1	103	101	98	2	150	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	4	85	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	5	100	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TB	1	103	101	98	6	150	101		0	0	1		0	0	0
									01	00	1		00	00	1
										0	0			0	0
											0				0
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*		RelPBS				SatPBS		
TH	1	103	101	98	0	1	98		1	0	0		0	0	0
									11	00	1		10	00	1
										0	0			0	0
											1				1

Figure 4.8: Tuples after attribute predicate type evaluation

fluct in Figure 3.11, the current predicate index that the evaluation will begin is at index 1, the second predicate in the list. If this predicate is satisfied, it indicates that a more covering predicate could be satisfied. Then the current predicate index is set to *current predicate index / 2*. If the predicate is not satisfied, the current predicate index is set to *current predicate index + predicate list size / 4*. The search is continued until the most covering predicate is found.

Returning to our example, predicate *T.fluct = 52.5* is not satisfied. But T's value is higher than the predicate value. This is an indication that the predicates in the upper half of the list could be satisfied. The only predicate in the upper half of the list is *T.fluct = 103*. If this predicate is satisfied, we have found the most covering predicate.

The binary search predicate evaluation method stops under two conditions. The first condition is if the evaluation reaches the beginning or end of the predicate list while trying to find the most covering predicate. The second condition is if

the evaluation is over a predicate list containing the '=' operator. The evaluation process ceases when the tuple has satisfied a predicate. There can be no covering predicate other than the satisfied one in the case of equality.

Figure 4.8 lists the tuples after the attribute predicate evaluation. Predicates corresponding to set BitSets in the tuples' RelPBS were evaluated and every predicate that was satisfied has the corresponding BitSet set in the tuple's SatPBS. The query list for each candidate tuple is re-evaluated using the attribute predicate of RelPBS and SatPBS. This updates the query list to only include queries that a tuple may satisfy after the attribute predicate evaluation.

Tuple TB with values (1, 103, 101, 98, 2l, 150, 101) had only one query ID, 2, in its query list. Query 2 requires the attribute predicate listed in BitPosition 1:1 to be satisfied. If the tuple's RelPBS in the corresponding BitPosition is also set, the predicate is relevant to the tuple and also has been evaluated. Therefore, if the tuple's SatPBS in the same BitPosition has not been set nor satisfied, the tuple no longer satisfies Query 2. This leaves TB's query list empty, indicating that this tuple will not satisfy any query, and it is removed from the intermediate buffer. The rest of the TB tuples have a similar evaluation.

Tuple TH has three query IDs in its query list. Each query will be evaluated one at a time. The first query, Query 1, requires that the predicate in BitPosition 0:0, *H.beatrate = 52*, be satisfied. The tuple's RelPBS in the corresponding BitPosition is also set, but the SatPBS in the corresponding BitPosition indicates that this predicate has not been satisfied. Thus, Query 1's ID is removed from TH's query list.

Query 2 requires predicate *T.fluct = 52.5* in BitPosition 1:1. This predicate is relevant to TH, but it has not been satisfied. Therefore Query 2 is also removed from TH's query list. Only Query 3 remains. Query 3 requires the predicate, *T.fluct*

$= 103$, in BitPosition 1:0 to be satisfied. Since the tuple has this BitPosition set in its SatPBS, Query 3 will remain a candidate query. Figure 4.9 lists the tuples that have survived the predicate type filtering. In this example, only one tuple remains.

									Query List
TH	1	103	101	98	0	1	98		3

Figure 4.9: Tuples' Query List after Second Predicate Type

4.3 Third Predicate Type Evaluation

For this running example, join is the last predicate type remaining to be evaluated. Evaluating the join predicate type is very similar to the evaluation of the window predicate type. Since this predicate type was not evaluated before the window predicate type, it is not going to use its exploitation structures to retrieve tuples to join. Therefore it will only be used as the last filter on the remaining tuples. Figure 4.10 lists the remaining tuple after the second predicate type evaluation.

Before the join predicate type is evaluated, tuple TH has its join segment of RelPBS set by its query, Query 3, contained in its query list. As shown in Figure 4.10, Query 3 requires only one join predicate in BitPositions 2:0 to be satisfied, which is $T.incr = H.vib$.

										Relevant				Satisfied	
									Attr	*Join*	*Win*		*Attr*	*Join*	*Win*
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*		RelPBS				SatPBS		
TH	1	103	101	98	0	1	98		1	0	0		0	0	0
									11	00	1		00	00	1
										1	0			0	0
											1				1

Figure 4.10: Tuples' RelPBS before Third Predicate Type Evaluation for WAJ Predicate Ordering

Join predicates are evaluated in a similar manner to the attribute predicates.

The join predicate evaluation will not be shown in detail, but the resulting SatPBS is shown in Figure 4.11. Tuple TH has satisfied Query 3's join predicate. This tuple will be forwarded to the final stage of this algorithm, which assesses if it has completely satisfied all the predicates for a given query before it is output. This way Juggler avoids outputting any false positives.

										Relevant				Satisfied	
									Attr	*Join*	*Win*		*Attr*	*Join*	*Win*
TH	*ts*	*fluct*	*deg*	*incr*	*ts*	*beatrate*	*vib*		RelPBS				SatPBS		
TH	1	103	101	98	0	1	98		1	0	0		0	0	0
									11	00	1		00	00	1
										1	0			1	0
											1				1

Figure 4.11: Tuples after Third Predicate Type Evaluation in WAJ ordering

4.4 Output Tuples

Before tuples are output to the queues, the tuples' validity to the queries in its query list are verified. For each tuple, two checks are conducted for each of the candidate queries before it is output. This is outlined in Figure 3.20. The first check is to compare the tuple's streams to the query's streams that were defined in the query's *From* clause. If the query's and tuple's streams are a match, the tuple proceeds to the second check. If not, more evaluation is needed to assess if the tuple is truly valid for this query. All the tuples for the missing stream in Juggler's WESs are retrieved to join before the tuples are processed by the second check.

The second check verifies that all the relevant predicates for a query have been evaluated before it is output to the query's output queue. The tuple's SatPBS is compared to the query's QED. If it is a superset, the tuple has evaluated all the relevant predicates and has also satisfied them. If the tuple's SatPBS is not a superset, the tuple's RelPBS is compared. If the tuple's RelPBS is a superset of

the query's QED, the tuple has not satisfied the required and relevant predicates of the query and the query's ID is removed from the tuple's query list. If the tuple's query list is empty, the tuple is removed from the list of output tuples. Otherwise, the tuple will repeat this verification process for its other queries.

If the tuple's RelPBS is not a superset, the tuple may still satisfy the query. This occurs when the tuple may have been missing required stream(s). For this case, the missing stream tuples were retrieved in the first check. The RelPBS and the query's QED are compared to assess if there are any predicates that are required by the query and were not previously relevant to this tuple. These predicates include the missing stream and therefore could not have been evaluated before. If so, these predicates are applied at this time and the tuple's SatPBS and QED are compared for a final verification. Note, when Predicate BitSets, such as the query's QED and tuple's JugglerPBS, RelPBS and SatPBS, are compared, the process is composed of bit operations, as described in Figure 3.15. After the tuples have been verified for all their queries, they are output to the queue.

In our example, only one tuple remains and before it is output, the candidate queries must be updated to denote satisfied queries. First, the streams for Query 3 are compared with the tuple's streams. Since the tuple contains all the streams for the query, there is no need to retrieve any tuples that may not have been retrieved during the evaluation phases. For example, if Query 3 required streams B, T, and H, tuples from Stream B's WESs would have been retrieved and joined with our intermediate tuple, TH. Then any predicates that involved stream B could not have been evaluated before. At this time, to remove any false positives, these predicates would be evaluated before the tuples are output.

In this example, tuple TH satisfies the stream requirement for Query 3 as stated within its *From* clause. Thus Query 3's QED will be used to compare the tuple's

SatPBS. If the SatPBS is a superset of Query 3's QED, the tuple is output. If not, the tuple is removed from the output tuple list.

In our example, only one tuple is output by the operator. This tuple only satisfies Query 3. The algorithm quickly identifies potential tuples, and also quickly filters ones that did not satisfy the queries. The ordering of the predicate types are key to reducing both the number of computations and the number of intermediate tuples.

4.5 Clean Up

Cleanup of the Juggler operator's exploitation structures occurs when an input tuple's timestamp requires a new WES bucket and the WESs' time range has exceeded the operator's maximum window size. This algorithm is described in Figure 3.22. In the running example, the creation of a new WES bucket was explained. If a tuple does not fall in the timestamp ranges of the WES buckets, a new bucket is created and the number of buckets in the WES is incremented. The start timestamp and end timestamp of the new WES is set to the tuple's timestamp value. The WES's time ranges also changes, which is the first bucket's first tuple's timestamp to the last bucket's last tuple's timestamp. As more tuples are entered in this bucket, the end timestamp of the bucket is updated. If the WES's time range is greater than the maximum window, the first bucket in the WES is purged. Copies of these tuples are also purged from the JES's Red-Black trees. This removes all tuples that have become stale in the operator.

After the input tuple T (1, 103, 101, 98) has been processed, it is entered into the exploitation structures for future processing with other tuples. To demonstrate the WES clean up process, we will assume more input tuples have arrived into the operator and have been processed. These tuples have been entered into WES and

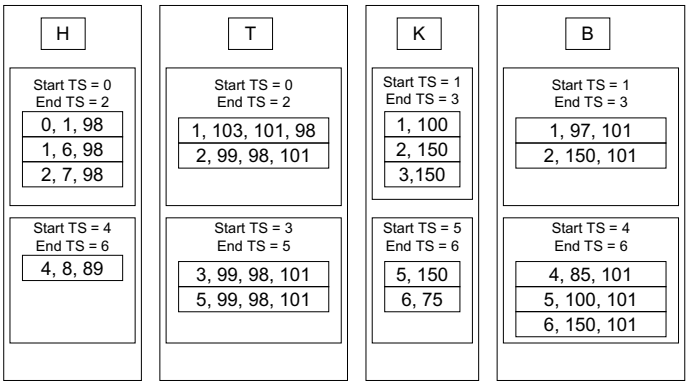

Figure 4.12: WES after processing input tuples

are shown in Figure 4.12. If another input tuple T (15, 99, 98, 101) now arrives into the operator, the first bucket's start and end timestamp in Figure 4.12 are compared to the input tuple's timestamp. Since the range is not included in this bucket, the next bucket is checked. Since none of the buckets include the input tuple's timestamp, a new bucket must be created. The range of the WES is 10, which is greater than the maximum window size of 6. This indicates that the tuples in the first bucket have most likely become stale, and all the tuples it contains are purged. The bucket IDs are decremented to indicate the purging of stale tuples, see Figure 4.13.

As each bucket is purged, each tuple in the bucket is removed from its relevant Red-Black trees. Since Red-Black trees do not store references, each Red-Black tree is considered and the each stale tuple is removed one at a time. This is not an optimal way of purging tuples from all of Juggler's data structures. Storing Java's WeakRef instead of copies of the tuples would be an option and should be investigated further.

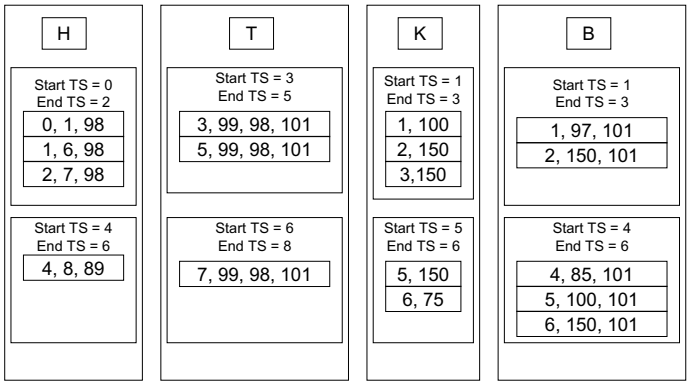

Figure 4.13: WES after purging stale tuples

4.6 Operator's Processing Logic

An operator is responsible for processing tuples, cleaning expired tuples, storing input and intermediate join tuples, and producing joined tuples in the output queue. The architecture and operator's processing logic is shown in Figure 4.14. To efficiently execute the operator's responsibilities, the operator's functionality consists of three modes. The first mode, Data Admission, filters tuples by applying the most selective type of predicate. In our example it is assumed to be the window predicate. The second mode, Propagation, applies the remaining two predicate types and stores the resulting joined tuples in the output buffer. Before a candidate tuple is output, the predicate bit encodings are compared with the queries' QEDs listed in the tuple's query list. This determines the tuple's relevancy to its queries. The last mode, CleanUp, maintains the window range by removing the expired tuples and updating the exploitation structures. Pseudo code of this functionality is described in Figure 4.15.

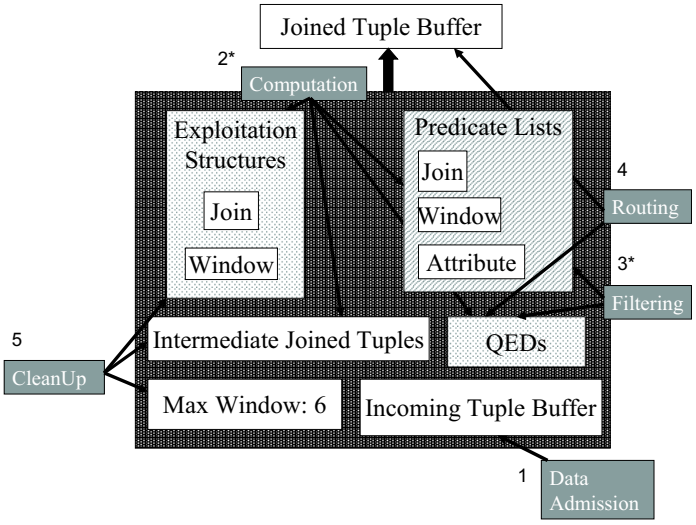

Figure 4.14: Architecture of an Operator

Mode 1: Data Admission

```
//Filter and buffer the new data
Check operator's most selective predicate type
 If tuple passes
    Append predicate bits
    Store in the operator's intermediate Buffer
```

Mode 2: Propagation

```
//Completes Predicates and Routes
Apply remaining 2 predicate types and
and set relevant predicate bits


//ROUTE
Route joined tuples to output buffer
```

Mode 3: CleanUp

```
//if a new data tuple exceeds
//the maximum time window boundary
Store tuple in exploitation structures, JESs and WES
Delete all tuples outside of window range
in the WES
        Also delete tuples in the Join Exp Structures
```

Figure 4.15: Juggler Operator's Functionality

Chapter 5

Assessing Juggler

5.1 Cost Model

The cost model estimates the cost of each operator in a query plan in terms of the number of intermediate tuples, the average cost of processing an input tuple, and the cost of accessing Juggler's data structures. It compares both the worst case scenario, which is a conservative estimate, and one simple example to display Juggler's savings in processing and space cost. The cost model is dependent upon the data structures in the Juggler operator as well as the algorithms chosen for this implementation.

5.1.1 Selectivity

Assuming a binary join of stream A and stream B, the operator AB has a window size which bounds the tuples' validity within a time. An operator can be shared by multiple queries, each contributing to the operator's local predicates. The operator contains the union of all the queries' predicates that share it. The operator's output tuples become the input for its parent(s).

$$T \rightarrow A\ window\ size \tag{5.1}$$

$$\lambda \rightarrow average\ arrival\ rate\ of\ stream\ (count/sec) \tag{5.2}$$

$$S\ =\ number\ of\ tuples\ in\ window\ for\ a\ stream(T\lambda) \tag{5.3}$$

The number of input tuples from one input/stream can be calculated using Equation 5.3. Assuming there are n input streams to the operator, the total number of input tuples entering into an operator is:

(5.4)

$$n\ is\ the\ number\ of\ inputstreams:$$

$$I_{op}\ is\ the\ number\ of\ the\ input\ into\ the\ operator$$

$$\lambda\ is\ arrival\ rate\ of\ stream\ i\ (count/sec)$$

$$I_{op}\ =\ \sum_{i=1}^{n}(\ T\lambda_i)$$

In Juggler, there are three types of predicates: attribute, window, and join. Each selectivity factor for a predicate type is calculated as the number of output tuples divided by the size of the Cartesian product of the input streams within a window size. This corresponds to all possible output tuples that could be produced if the predicates were not applied.

$$\sigma_j \rightarrow join\ selectivity\ factor\ for\ all\ join\ predicates\ in\ the\ range\ [0:1]$$

$$\sigma_w \rightarrow window\ selectivity\ factor\ for\ all\ window\ predicates\ in\ the\ range\ [0:1]$$

$$\sigma_a \rightarrow attribute\ selectivity\ factor\ for\ all\ attribute\ predicates\ in\ the\ range\ [0:1] \tag{5.5}$$

$$\begin{aligned} &The\ output\ of\ a\ traditional\ single\ functionality\ operator\ O_(op_{Trad}), \\ &M\ is\ the\ number\ of\ tuples\ in\ the\ operator\ to\ join\ with \\ &c_{op}\ is\ the\ number\ of\ all\ possible\ tuples\ output, \\ &Cartesian\ product\ of\ input\ tuples\ within\ a\ window\ size \\ &C_{op} = I_{op} \times M \\ &O_{op_{Trad}} \ = \ \sigma_a \ (C_{op}) \end{aligned} \tag{5.6}$$

The equations in 5.5 represent the selectivity factor of each predicate type. For example, assuming the possible number of output tuples in a traditional select operator is C_{op}, the number of tuples output from this operator would be equal to the selectivity of the attribute predicate multiplied by the size of the Cartesian product. This is shown in Equation 5.6. Similarly, if it had been a join operator, the selectivity of the join would have been applied to the size of the Cartesian product. Since Juggler applies all predicates to the input tuples, the number of output tuples for a given ordering in Juggler is shown in Equation 5.7.

$$O_{op_{wja}} \ = \ (\sigma_w)(\sigma_j)(\sigma_a)(C_{op}) \tag{5.7}$$

Notice that the selectivities of the three predicate types can be applied in any order resulting in the same output although the number of intermediate tuples would

differ as a consequence. This is what Juggler tries to exploit in its adaptive ordering of applying predicate types.

Assuming no operators are shared, the query plan is only composed of traditional single functionality binary operators. These operators combine selects and joins. Window predicates are join operators which join on the input's timestamp values. The number of tuples output for each of these single functionality operators can be easily calculated, as seen in Equation 5.8.

(5.8)

where a denotes an operator for an attribute predicate type

and s denotes an operator for a join predicate type

and w denotes an operator for a window predicate type

and $O_{qp_{Trad}}$ *is the output of the query plan with traditional binary operators*

$$O_{qp_{Trad}} = \sigma_{a_j} C_{op_j} + \cdots + \sigma_{a_k} C_{op_k} + \sigma_{j_s} C_{op_s} + \cdots + \sigma_{j_t} C_{op_t} + \sigma_{w_u} C_{op_u} + \cdots + \sigma_{w_v} C_{op_v}$$

$$O_{op_{Trad}} = \sum_{j=1}^{k} (\sigma a_j C_{op_j}) + \sum_{s=1}^{t} (\sigma_{j_s} C_{op_s}) + \sum_{u=1}^{v} (\sigma_{w_u} C_{op_u})$$

Adaptive Versus Static Predicate Ordering

Adaptive predicate ordering reduces the number of intermediate tuples that are processed in a query plan. Comparing an adaptive join algorithm with a static predicate ordering of JAW, <join, attribute, window>. We will assume from window of time of T_i to T_n, that predicates in the order selectivity will be JAW, namely join, attribute, window. From window time of T_{n+1} to T_m, we will also assume the data distribution has changed, and the order of predicate selectivity becomes AJW, <attribute, join, window>. Two ordering will be compared, static and dynamic.

Static predicate strategy will always apply predicates in JAW ordering. Dynamic predicate strategy will change its ordering to reflect the changes in data distribution. For window time of T_i to T_n, the dynamic predicate strategy will apply predicates in JAW ordering. For window time of T_{n+1} to T_m, the dynamic predicate strategy will adapt and apply predicates in AJW ordering.

The number of intermediate tuples after applying all three predicate types for time window T_i to T_n will be equal to the number of intermediate tuples for the adaptive predicate ordering. This is true for the case when the static predicate ordering is identical to the adaptive predicate ordering. The number of output tuples for this time is given in Equation 5.9. The number of the intermediate tuples after each predicate type evaluation is also given as Phase1, Phase2, and Phase 3. For time window T_i to T_n, Phase1 will represent the intermediate tuples resulting from applying the join predicate type. Phase2 will represent the intermediate tuples resulting from the attribute predicate type. Lastly, Phase3 will represent the intermediate tuples resulting from the window predicate type. For this time window, the number of intermediate tuples produced after each evaluation phase is the same for both static and adaptive predicate ordering.

(5.9)

$$for\ time\ 0 \leq T_i \leq T_n$$

$$for\ window\ time\ T_{s1} =\ T_i\ to\ T_n$$

$$attribute\ predicate\ type\ selectivity\ at\ window\ time\ T_{s1} is\ \sigma_{a_{s1}}$$

$$join\ predicate\ type\ selectivity\ at\ window\ time\ T_{s1} is\ \sigma_{j_{s1}}$$

$$window\ predicate\ type\ selectivity\ at\ window\ time\ T_{s1} is\ \sigma_{w_{s1}}$$

$$Cartesian\ product\ between\ the\ tuples\ within\ the\ window\ time\ T_{s1}\ is\ C_{op_{s1}}$$

$$with\ predicate\ ordering\ join,\ attribute,\ window\ (JAW)$$

$$O_{T_{s1}}\ is\ the\ output\ of\ the\ operator\ for\ the\ time\ window\ T_{s1}$$

$$O_{T_{s1}} = (\sigma_{w_{s1}}(\sigma_{a_{s1}}(\sigma_{j_{s1}}\ C_{op_{s1}})))$$

$$number\ of\ intermediate\ tuples\ at\ each\ predicate\ evaluation\ phase:$$

$$Phase1_{s1}\ is\ the\ output\ after\ the$$

$$first\ predicate\ type\ is\ applied\ for\ time\ window\ T_{s1}$$

$$Phase2_{s1}\ is\ the\ output\ after\ the$$

$$second\ predicate\ type\ is\ applied\ for\ time\ window\ T_{s1}$$

$$Phase3_{s1}\ is\ the\ output\ after\ the$$

$$third\ predicate\ type\ is\ applied\ for\ time\ window\ T_{s1}$$

$$Phase1_{s1} = \sigma_{j_{s1}} C_{op_{s1}}$$

$$Phase2_{s1} = \sigma_{a_{s1}} Phase1_{s1}$$

$$Phase3_{s1} = \sigma_{w_{s1}} Phase2_{s1}$$

The difference in the number of intermediate tuples between the two types of orderings can be seen for window size of T_i to T_n. In this time range, the selectivity of the predicate types has changed, resulting in the adaptive ordering of AJW, <attribute, join, window>. The static ordering remains JAW, <join, attribute, window>. Therefore, the number of intermediate results is listed in Equations 5.10. The intermediate results for the adaptive predicate ordering is listed in Equation 5.11.

(5.10)

$$for\ time\ T_n \leq T_{n+1} \leq T_m$$

$$for\ time\ T_{s2} = \ T_{n+1}\ to\ T_m$$

$$attribute\ predicate\ type\ selectivity\ at\ window\ time\ T_{s2} is\ \sigma_{a_{s2}}$$

$$join\ predicate\ type\ selectivity\ at\ window\ time\ T_{s2} is\ \sigma_{a_{s2}}$$

$$window\ predicate\ type\ selectivity\ at\ window\ time\ T_{s2} is\ \sigma_{a_{s2}}$$

$$Cartesian\ product\ of\ tuples\ within\ this\ time\ window\ T_{s2} is,\ C_{op_{s2}}$$

$$with\ predicate\ ordering\ join,\ attribute,\ window\ (JAW)$$

$$O_{T_{s2}}\ is\ the\ output\ of\ the\ operator\ for\ the\ time\ window\ T_{s1}$$

$$O_{T_{s2}} = \ (\sigma_{w_{s2}}(\sigma_{a_{s2}}(\sigma_{j_{s2}}\ C_{op_{s2}})))$$

$$number\ of\ intermediate\ tuples\ at\ each\ predicate\ evaluation\ phase:$$

$$Phase1_{static_{s2}}\ is\ the\ output\ after\ the$$

$$first\ predicate\ type\ is\ applied\ for\ static\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase2_{static_{s2}}\ is\ the\ output\ after\ the$$

$$second\ predicate\ type\ is\ applied\ for\ static\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase3_{static_{s2}}\ is\ the\ output\ after\ the$$

$$third\ predicate\ type\ is\ applied\ for\ static\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase1_{static_{s2}} \ = \ \sigma_{j_{s2}} C_{op_{s2}}$$

$$Phase2_{static_{s2}} \ = \ \sigma_{a_{s2}} Phase1_{static_{s2}}$$

$$Phase3_{static_{s2}} \ = \ \sigma_{w_{s2}} Phase2_{static_{s2}}$$

(5.11)

$$for\ time\ T_{s2}$$

$$with\ predicate\ ordering\ attribute,\ join,\ window\ (AJW)$$

$$O_{T_{s2}} = (\sigma_{w_{s2}}(\sigma_{j_{s2}}(\sigma_{a_{s2}}C_{op_{s2}})))$$

$$number\ of\ intermediate\ tuples\ at\ each\ predicate\ evaluation\ phase:$$

$$Phase1_{adaptive_{s2}}\ is\ the\ output\ after\ the$$

$$first\ predicate\ type\ is\ applied\ for\ adaptive\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase2_{adaptive_{s2}}\ is\ the\ output\ after\ the$$

$$second\ predicate\ type\ is\ applied\ for\ adaptive\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase3_{adaptive_{s2}}\ is\ the\ output\ after\ the$$

$$third\ predicate\ type\ is\ applied\ for\ adaptive\ strategy\ for\ time\ window\ T_{s2}$$

$$Phase1_{adaptive_{s2}} = \sigma_{a_{s2}}C_{op_{s2}}$$

$$Phase2_{adaptive_{s2}} = \sigma_{j_{s2}}Phase1_{adaptive_{s2}}$$

$$Phase3_{adaptive_{s2}} = \sigma_{w_{s2}}Phase2_{adaptive_{s2}}$$

Since the adaptive predicate ordering applies predicates in order of predicate type selectivity, the number of intermediate tuples passed to the next phase will be reduced. This is explained in more detail is Equations 5.12.

(5.12)

$$for\ time\ T_{s2}$$

$$output\ for\ both\ adaptive\ and\ static\ strategy\ is\ the\ same$$

$$adaptive\ strategy\ applies\ predicates\ in\ JAW\ ordering$$

$$dynamic\ strategy\ applies\ predicates\ in\ AJW\ ordering$$

$$number\ of\ output\ tuples\ of\ the\ adaptive\ strategy\ is\ O_{op_{ajw_{s2}}}$$

$$number\ of\ output\ tuples\ of\ the\ static\ strategy\ is\ 0_{op_{jaw_{s2}}}$$

$$O_{T_{s2}} = 0_{op_{ajw_{s2}}} = 0_{op_{jaw_{s2}}}$$

$$number\ of\ intermediate\ tuples\ at\ each\ predicate\ evaluation\ phase:$$

$$Phase1_{adaptive_{s2}} = \sigma_{j_{s2}} C_{op_{s2}}$$

$$Phase1_{static_{s2}} = \sigma_{a_{s2}} C_{op_{s2}}$$

$$since\ \sigma_{j_{s2}} < \sigma_{a_{s2}},$$

$$Phase1_{static_{s2}} > Phase1_{adaptive_{s2}}$$

$$after\ applying\ the\ second\ predicate\ type,$$

$$both\ orderings\ have\ applied\ the\ same\ predicate\ types:\ attribute\ and\ join$$

$$Phase2_{adaptive_{s2}} = \sigma_{a_{s2}}(\ \sigma_{j_{s2}} C_{op_{s2}})$$

$$Phase2_{static_{s2}} = \sigma_{j_{s2}}(\ \sigma_{a_{s2}} C_{op_{s2}})$$

$$Phase2_{static_{s2}} = Phase2_{adaptive_{s2}}$$

$$for\ the\ last\ predicate\ type,$$

$$Phase3_{adaptive_{s2}} = \sigma_{w_{s2}}(\ \sigma_{a_{s2}}(\ \sigma_{j_{s2}} C_{op_{s2}}))$$

$$Phase3_{static_{s2}} = \sigma_{w_{s2}}(\ \sigma_{j_{s2}}(\ \sigma_{a_{s2}} C_{op_{s2}}))$$

$$the\ number\ of\ intermediate\ tuples\ passed\ to\ this\ type\ are\ now\ identical,$$

$$Therefore,$$

$$Phase3_{static_{s2}} = Phase3_{static_{s2}}$$

In our simple comparison, we are only considering one operator at one point in time where the static predicate ordering is not optimal. In this example, $Phase1_{static} > Phase1_{adaptive}$ may not result in a significant difference. If more instances occurred where the ordering is not optimal in a static strategy, the difference in number of intermediate tuples for the two strategies widens. Therefore, with more instances of

data distribution changes, which changes the selectivity of predicates, the adaptive predicate ordering's cost savings will be more apparent.

Overlapping Predicates

Juggler contains Predicate Lists which group similar predicates in order to avoid repeated similar comparisons or predicate evaluations. In Juggler, predicates that have identical or similar structures are referred to as overlapping predicates. These are grouped to reduce the number of comparisons. We will use a simple example to compare the number of intermediate tuples for a query plan that contains non-overlapping predicates and another that contains two overlapping predicates.

```
Query 1:
Select *
From A, B
Window A.ts > B.ts
MaxWindow 2sec
Where A.col1 >= B.col2 and
      A.col1 > 50

Query 2:
Select *
From A, B
Window A.ts < B.ts
MaxWindow 2sec
Where A.col1 >= 2 * B.col2 and
      B.col1 < 40
```

Figure 5.1: Queries used to Compare Cost of Overlapping Predicates

Consider two query plans, each containing the two queries defined in Figure 5.1. The first query plan will contain one Juggler operator that will evaluate all the predicates of both queries. The second query plan we will consider contains two multi-functionality operators which evaluate both joins and selects, but it does not group similar predicates. The second query plan will contain two operators, each evaluating a query defined in Figure 5.1.

Since the window predicates are identical, and the attribute predicates are not similar, we will ignore the cost of computing these. Juggler will not be able to

exploit non-similar predicates. Just like Juggler, traditional operators can combine identical predicates. Therefore, the reduction in cost for these cases will not be unique to Juggler.

The join predicates of Queries 1 and 2, *A.col1 >= B.col2* and *A.col1 >= 2 * B.col2*, are similar since they share the same streams, columns, and operator. For the query plan that does not use Juggler's Predicate Lists, the join predicates of the two queries are evaluated in two operators. The cost of this is outlined in Equation 5.13. If there are tuples that satisfy both predicates, they will be duplicated in this query plan.

(5.13)

$$\begin{aligned}
&\textit{Assuming operator } Op_{1_{nonoverlap}} \textit{ contains Query } 1's \textit{ join predicate} \\
&A.col \ >= B.col2 \\
&\textit{and operator } Op_{2_{nonoverlap}} \textit{ contains Query } 2's \textit{ join predicate} \\
&A.col \ >= 2 \ * B.col2 \\
&\textit{and since both operators are have} \\
&\textit{the same maximum possible number of output tuples :} \\
&C_{op} = \ | \ C_{op_{1_{nonoverlap}}}| = |C_{op_{2_{nonoverlap}}}| \\
&\textit{and the selectivities of each join predicate is :} \\
&\textit{Selectivity of Query } 1's \textit{ join predicate } \rightarrow \sigma_{j_{Q1}} \\
&\textit{Selectivity of Query } 2's \textit{ join predicate } \rightarrow \sigma_{j_{Q2}} \\
&and \sigma_{j_{Q1}} < \sigma_{j_{Q2}} \\
&\textit{total number of intermediate tuples for query plan} \\
&\textit{with no overlapping predicates is } Total_{Inter_{nonoverlap}}
\end{aligned}$$

$$Total_{Inter_{nonoverlap}} = \sigma_{j_{Q1}}(C_{op_1}) + \sigma_{j_{Q2}}(C_{op_2})$$

$$Total_{Inter_{nonoverlap}} = (\sigma_{j_{Q1}} + \sigma_{j_{Q2}})((C_{op_1}) + (C_{op_2}))$$

$$Total_{Inter_{nonoverlap}} = (\sigma_{j_{Q1}} + \sigma_{j_{Q2}})(2 \times (C_{op})$$

In the Juggler operator, which uses the Predicate Lists, these two predicates will be grouped into one predicate list. When evaluating the data, a binary search evaluation will be used to find the most covering predicate. Since Juggler incorporates evaluating both predicates in one operator, the query plan only requires one copy of the input data. It also processes both predicates. The tuples that have satisfied the more covering predicate will then be processed for the other predicate, thus avoiding duplicating and repetitive evaluation of tuples which satisfy both predicates. Therefore, the Juggler operator reduces the cost in terms of number of intermediate tuples. This is shown in Equation 5.14.

(5.14)

$$\begin{aligned}
&\textit{Assuming operator } Op_{1_{nonoverlap}} \textit{ contains Query } 1's \textit{ join predicate} \\
&A.col >= B.col2 \\
&\textit{and also contains Query } 2's \textit{ join predicate} \\
&A.col >= 2 * B.col2 \\
&\textit{and the operator}'s \\
&\textit{maximum number of possible output tuples}: C_{op} \\
&\textit{and the selectivities of each join predicate is}: \\
&\textit{Selectivity of Query } 1's \\
&\textit{join predicate} \rightarrow \sigma_{j_{Q1}}
\end{aligned}$$

$$Selectivity\ of\ Query\ 2's$$

$$join\ predicate\ \rightarrow \sigma_{jQ2}$$

$$total\ number\ of\ intermediate\ tuples\ for\ the\ query\ plan$$

$$with\ overlapping\ predicates\ is\ Total_{Inter_{overlap}}$$

$$Total_{Inter_{overlap}} =$$

$$(\sigma_{jQ1}(C_{op})) + (\ \sigma_{jQ2}(\ \sigma_{jQ1}C_{op}))$$

$$Total_{Inter_{overlap}} =$$

$$\sigma_{jQ1}C_{op}(\ 1 + \sigma_{jQ2})$$

$$Total_{Inter_{overlap}} < Total_{Inter_{nonoverlap}}$$

$$Considering\ that\ duplicate\ tuples\ are\ not\ output,$$

$$the\ number\ of\ intermediate\ tuples$$

$$for\ an\ operator\ containing\ overlapping\ predicates,$$

$$Total_{Inter_{-Duplicates}}, is\ more\ reduced$$

$$Total_{Inter_{-Duplicates}} < Total_{Inter_{overlap}}$$

Multi-join Versus Binary join

Our overlapping predicate example simultaneously illustrates the reduction in the number of intermediate tuples when using a multi-join versus binary join operators. The input data had to be copied or reprocessed by each operator when using binary joins over the same data. Also similar to the overlapping predicates example, using a multi-join reduces the number of intermediate tuples, since tuples that have satisfied both predicates are not duplicated. The overlapping example indicated that combining multiple queries in one operator reduces the number of intermediate tuples in the query plan. If the operator accepted multiple inputs, the chance of

overlapping predicates in an operator increases. Thus the cost analysis of the overlapping example also explains the benefits of a multi-join over its equivalent binary joins.

5.1.2 Space Cost

The computation cost for each operator is a combination of the cost of processing the tuples and the size of Juggler's data structures. The cost of processing tuples incorporates cost of accessing and comparing the local predicates in the operator for each tuple. This cost is dependent upon the number of predicates registered in the operator and the number of predicates that are overlapping. These are predicates that are either identical or similar.

There are two data structures in the operator that we will consider, JES and Predicate Lists. WES is not considered since every operator needs to have a structure to store tuples within a time window. The variables used to calculate the space cost for all the predicates of registered queries in the operator are shown in the list below.

$$
\begin{aligned}
|j_p| &\rightarrow \text{total number of join predicates} \\
|w_p| &\rightarrow \text{total number of window predicates} \\
|a_p| &\rightarrow \text{total number of attribute predicates} \\
|n_p| &\rightarrow \text{total number of predicates}
\end{aligned}
$$

$$
|n_p| = |j_p| + |w_p| + |a_p| \tag{5.15}
$$

The maximum number of JESs in an operator is 2*$|j_p|$. This is the case where none of the join predicates share streams and columns. The maximum number of possible Predicate Lists is equal to the total number of the predicates registered in

the operator, Equation 5.15. This is the case when none of the predicates are similar and none could be grouped into a list. This situation also arises when each query is processed in separate operators or query plans without sharing any operators.

In the worst case, the size of Juggler's data structures will be as described above. Juggler will reduce the number of Predicate Lists and the number of JESs in the operator when there are two or more similar predicates. In the case when two join predicates share a stream and column, the number of JES in the operator is $2 * |j_p| - 1$. Each time a stream and column are shared in a join predicate, the number of JES required in the operator is reduced by 1.

Similarly, in the worst case, the size of the Predicate Lists in Juggler is equal to the number of registered predicates. When there are similar predicates, the number of Predicate Lists in the operator is reduced by 2 since there are two streams and columns that are similar to two other predicate's streams and columns. In the best case scenario, when all predicates are similar for all three predicate types, the minimum number of Predicate Lists required is three. In this case, Juggler will contain one predicate list for each type containing all the predicates of that type.

Therefore, the more predicates that overlap the more number of JESs and Predicate Lists in Juggler decreases. Note the cost of these structures is specific to Juggler's implementation. In this analysis, the cost of access and cost of storing tuples into the index has not been considered. Currently, Juggler implements Predicate Lists as ArrayLists. The difference in cost of using a different implementation structure such as a hash table should be compared. This cost is an overhead for Juggler's attempt to exploit similar predicate evaluation and also to quickly retrieve tuples to join. The operator's overhead cost can be compared to other operator implementations with similar functionality. This is left for future work.

Processing Cost

The processing cost of a tuple is dependent upon the number of predicates registered in the operator and the data structures and the number of overlapping predicates. Similar predicates will reduce the size of the Join Exploitation Structure and Predicate Lists and thus reduce the cost of processing the tuples. We will assume that all predicates are non-overlapping unless otherwise stated.

To process a tuple in a Juggler operator, few factors determine the processing cost: cost of binary search over the Attribute Predicate Lists, cost of binary search over the Join Predicate Lists, cost of accessing tuples to join from the JESs, cost of binary search over the Window Predicate Lists, and the cost of retrieving tuples to join from the WESs. This is shown in Equations 5.16. First we will assume the worst case, where there are no overlapping predicates.

In the worst case scenario, the number of predicate lists for each type is equal to the number of predicates for that type. The cost of accessing a predicate is equal to a sequential search over each Predicate List, O(n), where n is the size of the list. Since all the predicates are non-overlapping and cannot be grouped, all Predicate Lists are of size 1. Therefore, the cost of accessing each predicate is 1 and the cost of accessing all the predicates is equal to the number of predicates. The cost of accessing each predicate in this scenario can be represented as:

$$C_i \rightarrow \textit{cost of traversing a predicate list} \tag{5.16}$$

Cost of accessing a predicate within the predicate list is:

$$\begin{aligned} C_{jp} &\rightarrow \text{cost of accessing join pred} &&= |j_p| * C_i \\ C_{wp} &\rightarrow \text{cost of accessing win. pred} &&= |w_p| * C_i \\ C_{ap} &\rightarrow \text{cost of accessing attr. pred} &&= |a_p| * C_i \end{aligned}$$

In the worst case, cost of accessing a predicates, is $C_i = 1$

$C_p = |j_p| + |w_p| + |a_p|$

If there is one similar join predicate, the operator contains $|j_p| - 1$ join Predicate Lists. The cost of the binary search over this predicate list is $O(\log(|j_p|))$, which in this case will equal 1 since the size of the list is 2. Therefore, the cost of accessing each join predicate will become $C_i(|j_p| - 1) + 1$. Therefore, as the number of similar predicates increases in an operator, the more it reduces the cost of accessing a predicate to evaluate.

Similarly, the cost of retrieving tuples from the JES is represented by C_{JES}. In the worst case scenario, the cost of evaluating all the join predicates is to use half of the JESs to retrieve tuples to join, $(1/2)(|j_p|C_{JES})$.

In the case where there are two similar join predicates, the number of JESs in the operator becomes $|j_p| - 1$. If all predicates are relevant to a tuple, half of the JESs will be used to retrieve tuples that satisfy each predicate. Thus, cost of retrieving tuples from the JESs is $1/2(|j_p| - 1)C_{JES}$, which less than the worst case scenario. As more similar predicates are registered in a Juggler operator, the cost of retrieving tuples from the JESs are reduced. Cost of retrieving tuples from the WESs has a similar cost analysis as the analysis of JESs.

Cost of processing a tuple is composed of binary searches over the Predicate Lists to find the most covering and the cost of retrieving tuples to join. In the worst case scenario, for each of the subparts of the cost calculation represents single functionality operators. Juggler's cost analysis indicates that Juggler exploits similar predicates and thus reduces the cost of processing an input tuple.

Even though the cost of processing an input tuple is reduced when using a

Juggler operator, the overhead of the data structures needs to be determined. Implementing Juggler with less memory intensive and time consuming data structures will highlight Juggler's savings. The current implementation uses memory and time intensive structures, which may overshadow Juggler's savings in processing input data.

5.2 Experimental Evaluation

Several experiments were run to assess if Juggler's mega operator processes multi-way joins with numerous predicates. Experiments were conducted to assess if Juggler adapts its predicate ordering to the dynamic characteristics of data streams. Several different environment variables, such as stream statistics, number of registered queries, and number of input streams, will determine predicate ordering. Data was generated using CAPE's Data Generator and was streamed into the operator using CAPE's Stream Generator. The input streams, data rates, and values are synthetically generated by the system.

To test the features of the proposed system, I ran the following tests:

1. Compare the total number of output tuples of a plan composed of one multi-join versus its corresponding plan composed of several binary join operators.

2. Compare adaptivity of the predicate ordering vs. a static ordering for the same query and data set.

3. Compare the number of comparisons required for the operator which adaptively reorders predicate types vs. static predicate type ordering.

4. Identify system limitations in terms of number of streams, queries, and predicates our composite operator can handle.

5.2.1 Binary versus Multi-joins

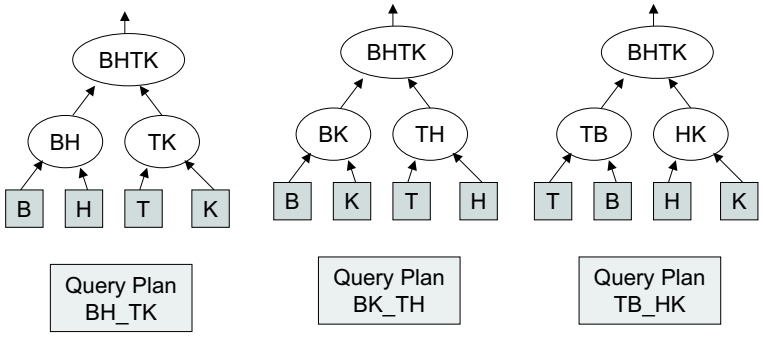

Figure 5.2: Query Plans with Binary Operators

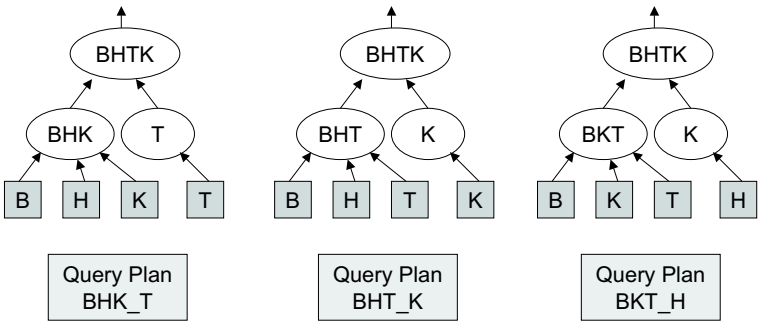

Figure 5.3: Query Plans with Multi-join Operators

Tests were run ten times each on an IBM T40 laptop with 1 Gigabyte of RAM. The average statistics were used to assess the performance of Juggler's features. These tests revealed a drawback of Juggler in its current implementation: Juggler is very time consuming. Cleanup of Juggler's data structure, cost of traversing the Juggler tuple representation, avoiding duplicate joined tuples, and converting Juggler tuples to CAPE's XATTuples and vice versa are all memory and time intensive.

To compare the total number of output tuples for a multi-join versus its corresponding binary joins, four inputs with 100 tuples in each input were processed over several equivalent query plans. Some query plans were composed of several binary

joins and other query plans were composed of a combination of a multi-join and a binary join operator. BH_TK, BK_TH, and TB_HK are query plans composed of binary joins and are shown in Figure 5.2. The remaining query plans are multi-joins combined with a binary join as shown in Figure 5.3 and are named BHT_K, BKT_H, and BHK_T.

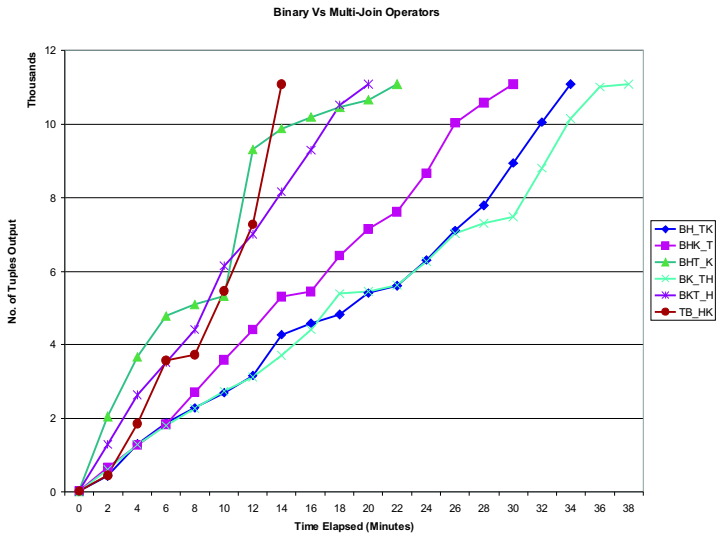

Figure 5.4: Binary and Multi-join Query Plans

Figure 5.4 charts the output of each query plan over time. One binary query plan, BH_TK, had the fastest output rate. Although, on average, the output rate of the multi-join query plans was better than the output rate of the binary plans. This is promising indicating that multi-join operators can lead to faster output than their equivalent binary joins. Overall, the operator output rate was very slow due to the current non-optimized implementation of Juggler's structures. To assess the cause of this performance hit, the time spent on traversing the tuple's linked list structure was measured. This time consisted of either retrieving a tuple's value or avoiding creating duplicates. Figure 5.6 compares the time taken for each query plan to finish processing and the time taken to traverse the JugglerTuple structure.

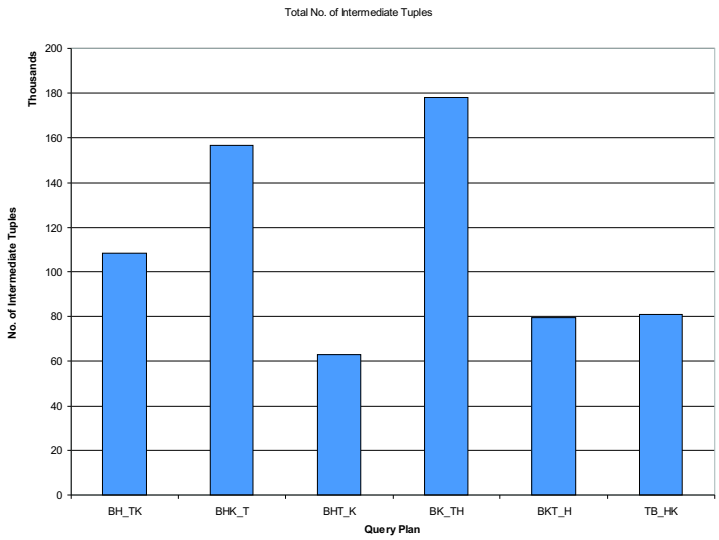

Figure 5.5: Number of Intermediate Tuples for Binary and Multi-join Query Plans

According to Figure 5.6, JugglerTuple's current implementation spends a lot of time in traversing the data structure, and that this would be an area to investigate for future improvements.. More than fifty-percent of the processing time was taken to traverse the JugglerTuple structure. The same figure also indicates that most multi-join query plans either reduced processing time by a considerable amount or performed comparably to the binary join operators. Replacing the time intensive structures with more optimal structures can further reduce processing time.

The number of intermediate tuples for each query plan was also measured. This is the sum of the intermediate tuples resulting after each operator's predicate type evaluation. Figure 5.7 indicates that most multi-join query plans significantly reduced the number of intermediate tuples. One binary plan, TB_HK, had a reduced number of intermediate tuples, which was not the case with other binary plans. As the size of intermediate tuples in Juggler increased, JugglerTuple, a tuple structure used in Juggler, consumed more time, resulting in a slower output rate, as seen in BK_TH's performance. Juggler implements a joined tuple as a linked list of Juggler

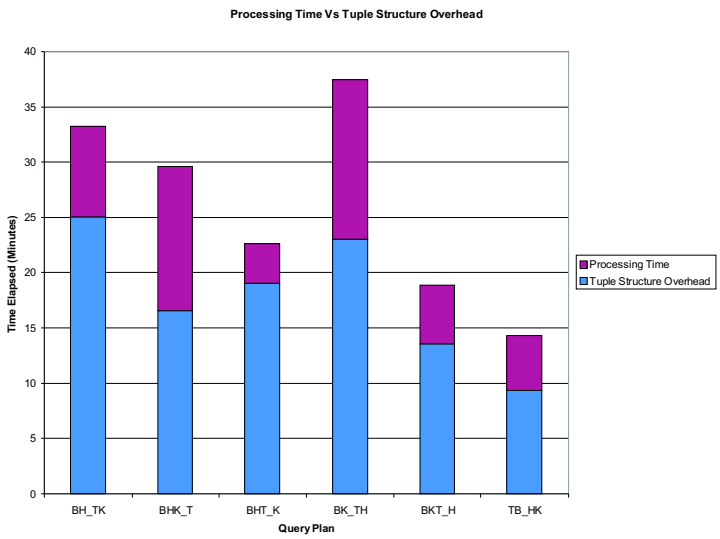

Figure 5.6: Tuple Overhead for Binary and Multi-join Query Plans

Tuples. Avoiding duplicates and joining tuples require traversal of a tuple's linked list and this greatly compromised performance. Overall, query plans with multi-join operators had a faster output rate, seen in Figure 5.4, and also greatly reduced the number of intermediate tuples produced within the operators.

5.2.2 Predicate Reordering

To compare the adaptivity of the predicate ordering, one query was registered into the operator. The query plan for this test is shown in Figure 5.9. The query is listed in Figure 5.8. Tests were run while the operator dynamically reordered the evaluation of the predicate types. The same query was run with each of the six static predicate orderings. Figure 5.10 charts the performance of each predicate ordering. The names of the query plans are followed by three letters to indicate the predicate ordering. For example, AJW indicates the static predicate ordering of Attribute-Join-Window. The query plan with no predicate ordering following its

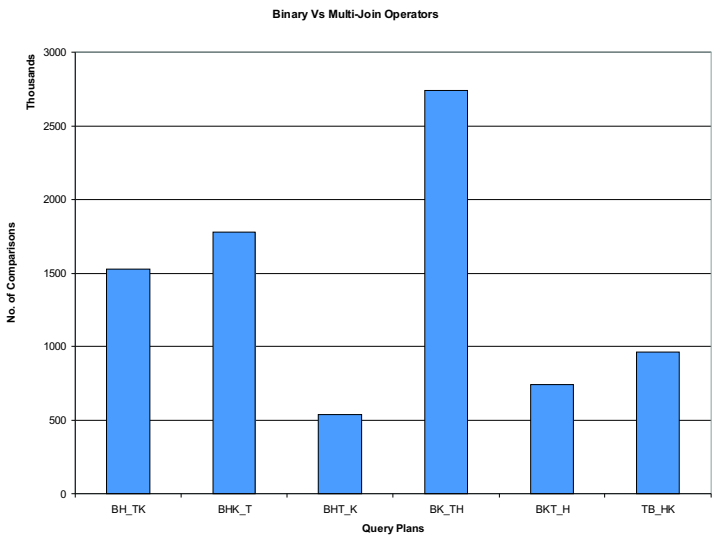

Figure 5.7: Number of Comparisons for Binary and Multi-join Query Plans

```
Select *
From B,H,T
Window B.ts <= T.ts and H.ts <= T.ts
MaxWindow 20sec
Where B.pressure < 90 and
        H.beatrate > 115 and
        B.pressure <= H.vib and
        B.ts <= H.ts and
        T.fluct lt 200 and
        B.temp lt= T.degree and
        H.vib lt= T.fluct
```

Figure 5.8: Query used to test Predicate Reordering

name dynamically reorders the predicate types. For this particular query and its stream statistics, the dynamic predicate reordering fared as well as the static ordering of AJW. There is a point when the dynamic reordering had a higher output rate, but the static AJW quickly caught up. More tests with varying stream statistics and queries must be run to formulate a concrete claim.

The bar chart in Figure 5.11 illustrates the number of comparisons the static predicate orderings incurred versus the number of the comparisons incurred by the dynamic predicate orderings. The bar chart indicates that the adaptive predicate

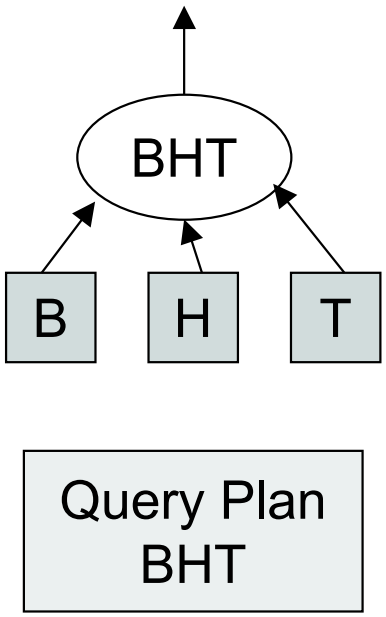

Figure 5.9: Query Plans with One Multi-join Operator

reordering results in fewer comparisons than all the other static orderings. A comparison is defined as any calls comparing BitSets, exploitation structures, and evaluation of predicate lists. The number of comparisons decreases when using adaptive reordering. Candidate tuples are discarded earlier in the join algorithm if the evaluation of predicate types adapts to changing data distributions. For this particular query, the attribute predicate type was consistently the more selective predicate type. Hence, it performed comparably to the dynamic reordering test. The cost of bit comparison is not as expensive as a lookup in the WES. To completely assess the benefit of reducing the number of comparisons, more tests are needed in order to identify the specifics of these comparisons.

5.2.3 Overlapping Predicates

Another set of experiments was run to assess if overlap of similar predicates in multiple queries improved performance. Five queries were registered into each query plan

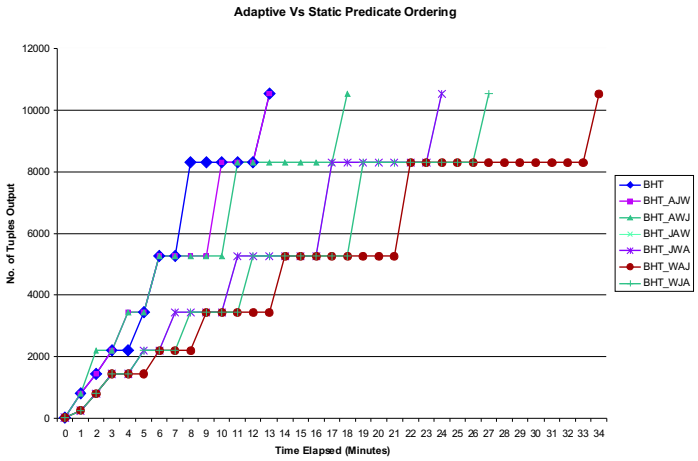

Figure 5.10: Adaptive vs. Static Predicate Orderings

containing one Juggler operator, BH. These queries are shown in Figures 5.13, 5.14, and 5.15. The first query plan contained approximately twenty-five percent overlap of predicates. Predicates are considered overlapping when they can be grouped into one predicate list. These predicates can be identical or similar. The second query plan contained approximately fifty percent overlap and the third contained approximately seventy-five percent overlap. When an operator contains overlapping predicates, many of the predicates can be evaluated using the binary search on the predicate lists. This reduces the number of predicate lists in the operator and also reduces the number of comparisons required to process each input tuple.

The results were promising and indicates that the overlap of predicates were exploited by Juggler. There was a consistent percentage of decrease in the number of comparisons required when queries had a greater percentage of overlapping predicates as depicted in Figure 5.12.

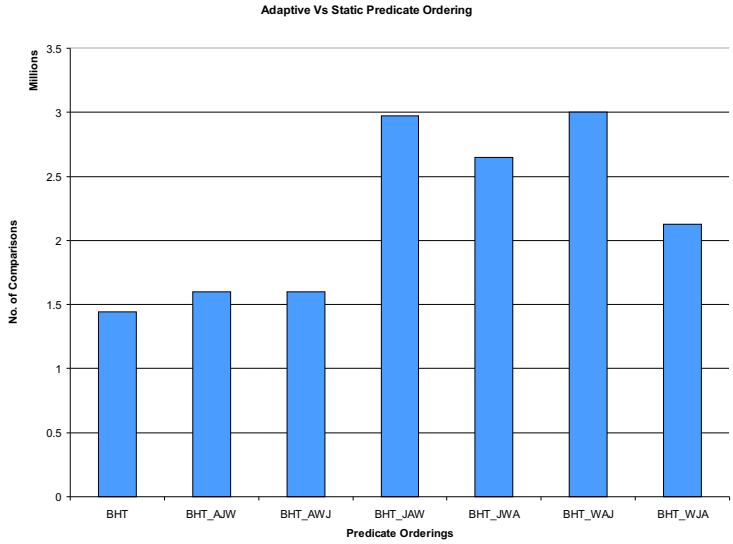

Figure 5.11: Number of Comparisons for Predicate Orderings

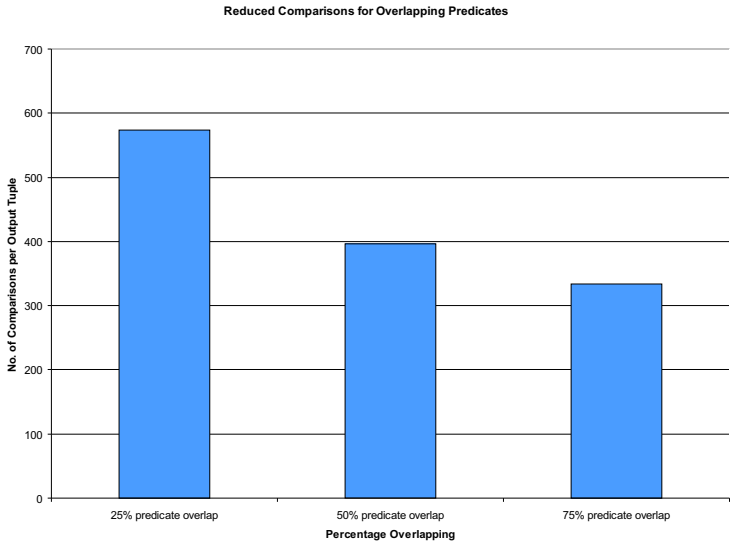

Figure 5.12: Queries with Overlapping Predicates

Twenty-Five Percent Overlap

```
Query 1:

Select *
From BH
Window  B.ts > H.ts
Max Window 20 sec
Where   B.pressure < 90 and
        B.pressure > H.vib


Query 2:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 115 and
        H.beatrate > 115 and
        B.pressure <= H.vib * 2 and
        B.pressure < H.beatrate


Query 3:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 116 and
        H.beatrate >= 115 and
        H.vib > 80 and
        B.pressure <= H.vib * 3 and
        B.pressure > H.beatrate and
        B.temp > H.beatrate


Query 4:

Select *
From BH
Window  B.ts gt H.ts
Max Window 20 sec
Where   B.pressure < 117 and
        H.beatrate < 115 and
        H.vib < 80 and
        B.temp > 90 and
        B.pressure <= H.vib * 4 and
        B.pressure = H.beatrate and
        B.temp >= H.beatrate and
        B.temp > H.vib


Query 5:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure < 118 and
        H.beatrate = 115 and
        H.vib = 100 and
        B.temp = 98 and
        B.pressure > H.vib and
        B.pressure >= H.beatrate and
        B.temp < H.beatrate and
        B.temp >= H.vib
```


Figure 5.13: Query 1 Predicate Overlap

Fifty Percent Overlap

```
Query 1:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure < 110 and
        B.pressure <= H.vib


Query 2:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure > 113 and
        H.beatrate > 110 and
        B.pressure >= H.vib and
        B.pressure > H.beatrate


Query 3:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure > 115 and
        H.beatrate >= 110 and
        H.vib gt 80 and
        B.pressure >= H.vib  * 2 and
        B.pressure < H.beatrate and
        B.temp > H.beatrate


Query 4:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure > 117 and
        H.beatrate < 110 and
        H.vib < 80 and
        B.temp > 90 and
        B.pressure >= H.vib * 3 and
        B.pressure = H.beatrate and
        B.temp >= H.beatrate and
        B.temp >= H.vib


Query 5:

Select *
From BH
Window  B.ts >= H.ts
Max Window 20 sec
Where   B.pressure > 120 and
        H.beatrate = 115 and
        H.vib = 100 and
        B.temp = 98 and
        B.pressure >= H.vib * 4 and
        B.pressure >= H.beatrate and
        B.temp < H.beatrate and
        B.temp <= H.vib
```


Figure 5.14: Query 2 Predicate Overlap

Seventy-Five Percent Overlap

```
Query 1:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 110 and
        B.pressure <= H.vib and


Query 2:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 113 and
        H.beatrate > 115 and
        B.pressure <= H.vib and
        B.temp > H.beatrate


Query 3:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 115 and
        H.beatrate > 117 and
        H.vib > 80 and
        B.pressure <= H.vib * 2 and
        B.temp > H.beatrate * 2 and
        B.pressure < H.beatrate


Query 4:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 117 and
        H.beatrate > 120 and
        H.vib < 80 and
        B.temp > 90 and
        B.pressure <= H.vib * 3 and
        B.temp > H.beatrate * 3 and
        B.pressure = H.beatrate and
        B.temp >= H.vib


Query 5:

Select *
From BH
Window  B.ts <= H.ts
Max Window 20 sec
Where   B.pressure < 120 and
        H.beatrate > 125 and
        H.vib = 100 and
        B.temp = 98
        B.pressure <= H.vib * 4 and
        B.temp > H.beatrate * 4 and
        B.pressure >= H.beatrate and
        B.temp = H.vib
```


Figure 5.15: Query 3 Predicate Overlap

Chapter 6

Conclusion

In the area of real time streaming data, continuous queries are used to process streaming data from multiple sources into results useful for various fields. From medicine to the stock market, if the data becomes stale, it is no longer useful. The nature of streaming data also makes it impossible to get a complete result. Intermittent outputs are necessary.

Juggler, a multi-join operator, proposes a solution in the CQ area. With the use of multiple query plans, joins, and selects in one operator, Juggler bounds the streaming data using window joins and a maximum window size.

The Juggler operator combines three features into one operator. The contributions of Juggler are:

- Grouping similar predicates.
- Reordering predicate evaluation.
- A multi-join operator with multiple attribute and join predicates.

The feature of Juggler which promises to be the most useful is the dynamically reordering and applying the predicate types in order of selectivity, allowing Juggler

to dynamically adapt to the changing data stream distribution. Although the area of continuous queries has been investigated by many, Juggler's approach of dynamically ordering predicate types in a multi-join mega operator has not been done.

Juggler has shown to reduce computations when predicates overlap. This improvement is the result of Juggler's grouping of similar predicates. Efficiencies also come into play by dynamically changing the order of predicate type evaluation in a multi-join operator. Multi-join operators collapse a query plan or a subset of a query plan into one operator. If a multi-join operator dequeues data from inputs in a predefined order, the operator may block if an input has no data. Juggler will dequeue from any input that has data to process. This avoids blocking an operator from waiting on a slow input stream, instead Juggler will proceed to process tuples from other inputs.

The Juggler operator is a mega operator which proposes a solution to processing continuous queries with sliding windows. Juggler has three contributions: it proposes an adaptive predicate type reordering mechanism, it groups similar predicates in an attempt to reduce the number of comparisons, and it combines joins and selects into one mega multi-input operator. Juggler proposes an adaptive join algorithm by reordering predicate type evaluation within the operator. Tuples are routed through different predicate types to adapt to changing stream characteristics.

Juggler was designed and implemented in Java. Preliminary tests were conducted to probe Juggler's features and performance. On average, multi-join outperformed its equivalent binary joins. Adaptive predicate ordering resulted in a reduction in the number of comparisons when queries contained many similar predicates. More experiments are needed to confirm and identify the limitations of these features.

6.1 Future Work

Much work remains for Juggler. The most important remaining task is optimization. When Juggler was integrated into the CAPE system, tuples were converted and reverted to CAPE's format of tuples. This caused some performance degradation. Also, Juggler's tuple representation is an extended form of Linked List. This also resulted in a performance hit, since duplicate elimination was very expensive with this tuple design. Since JugglerTuples are implemented as a LinkedList, to identify a duplicate tuple, each tuple's linked list is traversed and the tuples' values and timestamps are compared. This is explained in more detail in the appendix. The WESs are also implemented as linked lists, and thus another factor in the performance hit. JugglerTuples can be converted to an extending class of CAPE's XATTuples which are implemented as an array of values instead of a LinkedList. This would simplify duplicate elimination and retrieval of a tuple's value.

6.1.1 Adaptive Predicate Reordering

More research in the timing and frequency of predicate reordering is needed. Currently, Juggler reorders predicates after a pre-defined set of tuples have been processed, which may not be an optimal solution. Instead, predicate type selectivities can be updated when the operator can detect a significant change in data distributions. This would require another algorithm that monitors streams' statistics.

6.1.2 Policies

More research is also needed in the output policy. Currently, Juggler outputs tuples as they are processed.. Approximate output may be acceptable in certain applications. A more flexible policy may be preferable, where the user can define if an

approximate or complete result policy is desired.

6.1.3 Optimization

The current implementation of Juggler did not investigate the cost and overhead of its implementation choices and structures. As seen by the experimental results, Juggler's Tuple structure was very time-consuming which affected its output rate. This is only one of many data structures which need optimizing. Also, there may be some redundant bit comparisons during processing. Further investigation of the algorithm can identify these areas. Some data structure that will need optimizing are listed below.

- The Juggler tuple representation is a LinkedList.
- The duplicate elimination process traverses Linked Lists.
- The WES buckets use the Juggler tuple's representation.
- JESs store tuple copies.

6.1.4 Performance

As stated in the Experimental Evaluation section, more tests are needed to assess Juggler's features. Preliminary tests probed and indicated some initial results, but Juggler's implementation limitations were also identified, in particular its scalability. Work is needed to optimize data structures to determine the limitations and performance of Juggler and to evaluate the maximum number of input streams and queries it can handle.

Juggler incorporates innovative ideas and proposes a possible solution in a dynamic, bursty, and real-time environment of CQ. Juggler proposes one possible

solution in this relatively new and uncharted research area. It combines three novel ideas of reordering predicate types, grouping similar predicates, and combining joins and selects into one operator. This has indicated to be promising solution.

Bibliography

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 261–272. ACM, 2000.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, 2002.

[3] S. Babu and J. Widom. Continuous Queries over Data Streams. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA*, volume 30, pages 109–120, 2001.

[4] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 379–390. ACM, 2000.

[6] P. Haas and J. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. ACM Press, 1999.

[7] M. Hammad, W. Aref, and A. Elmagarmid. Joining multiple data streams with window constraints. *Proceedings of the 28th VLDB Confrerence*, 2002.

[8] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[9] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins Over Unbounded Streams. http://www.cs.wisc.edu/niagara/papers/knv02-windowjoin.pdf, 2002.

[10] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2002, Madison, Wisconsin, USA*, pages 49–60. ACM, 2002.

[11] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM Press, 2002.

[12] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Conference on Cooperative Information Systems*, pages 162–173, 2000.

[13] J. Ullman and J. Widom. A first course in database systems, 1997.

[14] T. Urhan and M. J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, 1999.

[15] S. Viglas, J. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. http://www.cs.wisc.edu/niagara/papers/mjoin.pdf, 2002.

[16] S. Viglas, J. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002.

[17] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 4-6, Madison, Wisconsin, USA*, 2002.

Appendix A

Juggler Implementation

A.1 Juggler Operator

PURPOSE: XATJugglerOperator is an interface to the CAPE system. It allows the operator to run and initialize its state. XATJugglerOperatorImp implements this interface and processes the input tuples and initializes the queries and their predicates.

The Juggler operator extends the XATMultiQueueStreamOperator which extends both the XATMultiSourceOperator and the XATStreamOperator. Three methods in XATJugglerOperatorImp are used to initialize and run the operator. One method is visitInt. Its parameters are: String numQueues, String queryInfo, String attrPredicates, String joinPredicates, and String winPredicates. This method is called by edu.wpi.cs.dsrg.xmldb.xat.component.queryplangenerator.ClassVariableVisitor during the query registration and query plan generation. The method initializes the operator's queries and their predicates in Juggler's structures. After this method, the predicates need to be parsed and stored in Juggler's predicate representation structures.

When the operator is given a chance to run by CAPE's Execution Controller, the operator checks to see if its state has been initialized. If not, the operator's initialize method is called. This method parses the predicate strings that were passed to visitInit. As predicates are parsed, the predicate lists are also created and entered into the appropriate lists followed by the creation of . the queries' QEDs. The operator then registers itself with the input queues and output queue. Also the operator's three predicate managers, attribute, join and window, are initialized by creating predicate lists and exploitation structures. After the initialize method has been called, the operator is prepared to process tuples. If the operator has been initialized, the operator dequeues tuples and calls its PredicateOrderingManager to process them.

Before each predicate type is applied, the tuples are converted into JugglerTuples. JugglerTuple extends XATTuple and contains a structure, Join Predicate BitSet Structure (JugglerPBS), that is used by the operator to filter the intermediate tuples. The PredicateOrderingManager processes tuples by applying predicate types in order of selectivity. Each predicate type manager applies predicates and the intermediate tuples are filtered by comparing the tuple's JugglerPBS to the queries' QEDs. The intermediate tuples that remain after all three predicate types are applied are tuples that are output. Before tuples are output, the tuples are entered into the operator's exploitation structures and then are re-converted to XATTuples. The structures specific to the Juggler operator, such as JugglerPBS, are only relevant to the operator that created it.

A.2 Predicate Ordering Manager

PURPOSE: The Predicate Ordering Manager is responsible for processing and entering the tuples into the operator's structures. It contains the three predicate type managers: attribute, join and window. Tuples are passed to each predicate manager in order of predicate type selectivity. It also maintains selectivity of each predicate type and reorders them to reflect changes in data characteristics. After joining tuples using Juggler's join algorithm and checking to confirm they have satisfied at least one query, the list of resulting joined tuples is returned to XATJugglerOperatorImp.

XATJugglerOperatorImp calls the PredicatOrderingManager's processTuples method. This method incorporates Juggler's processing algorithm outlined in Figure 3.19. This class has many helper methods for processing tuples, such as creating a tuple's JugglerPBS before processing, comparing tuples' RelPBS to query QEDs, and maintaining selectivities of each predicate type.

A.3 Predicate Type Manager

PURPOSE: There is a predicate type manager for each predicate type. Join and window predicate type managers contain a collection of Predicate Lists and exploitation structures. Attribute predicate manager only contains Predicate Lists. The Predicate Lists are populated when the PredicateOrderingManager's initialization method is called. Each manager must be able to find the most covering predicate for each predicate list it contains, retrieve relevant tuples to join when given an input tuple, and apply all relevant predicates to an intermediate tuple. Each predicate type managers also contains its selectivity .

Each of the three predicate type managers implements IPredicateManager, Figure A.2. This interface requires all its classes to implement two methods: getSe-

lectivity() and applyPredicates(). This interface allows PredicateOrderingManager to reorder the predicate type by selectivity and call each type manager's method to process tuples.

Each of the predicate managers implements the method registerQueries(). This method is called during the Juggler operator's initialize method. Each predicate manager creates the predicate lists and associates each predicate to its queries.

Attribute Predicate Manager differs slightly from the Join and Window Predicate Managers. JWPredicateManager is an interface that extends IPredicateManager. It incorporates the differences between attribute and the other two predicate types. These differences are reflected in the predicate structure, predicate lists, and exploitation structures. Attribute predicates only have one side that involves a stream's value and the other side is a constant. Join and Window predicates have both sides that involve stream values. Attribute predicate manager does not contain any exploitation structures but Join and Window predicate managers contain exploitation structures, JES's Red-Black trees, and WES's buckets respectively.

A.3.1 Attribute Predicate Manager

PURPOSE: Attribute predicate manager finds the most covering predicates from its Predicate Lists and also applies only the relevant predicates to an intermediate tuple.

Attribute predicate manager does not contain any exploitation structures. PredicateManager is a class that extends IPredicateManager. It contains AttrPredOrderListS, which is a structure that orders the attribute predicates. This structure will be described in more detail later, but it assumes that the attribute predicate has three parts: stream value, an operator, and a constant.

The method, applyPredicates(), takes an Iterator and an input tuple. If this

predicate type is the first or most selective predicate to process tuples, the Iterator will be null. In this case, the input tuple will have all the attribute predicates applied. Otherwise, the Iterator can be a collection of intermediate tuples and the manager will only evaluate each tuple's relevant predicates.

A.3.2 Join Predicate Manager

PURPOSE: Join predicate manager finds most covering predicates from its Predicate Lists, retrieves relevant tuples to join from its JESs and applies only the relevant predicate to intermediate tuples.

JWPredicateManger is an interface that contains an array of Red-Black trees implemented by java.util.TreeMap. It also contains a list of OrderListS, which differs from AttrPredOrderListS. Join predicates, as mentioned above, differ from attribute predicates in their structure. OrderListS incorporate this difference and contains these predicates in an ordered manner.

Join Predicate Manager also implements the method applyPredicates(). The parameters are also the same as the parameters described in the Attribute Predicate Manager. If this predicate type is the first type to process the tuples, the Iterator is null. In this case, all the predicates are applied to the input tuple. If the Iterator is not null, only the relevant predicates are applied to the intermediate tuples.

A.3.3 Window Predicate Manager

PURPOSE: Same as Join Predicate Manager.

Similar to Join Predicate Manager, Window Predicate Manager also differs from the Attribute Predicate Manager for the same reasons. It also extends JWPredicateManager, but the Window Predicate Manager will initialize the array of JES in the JWPredicateManager to null, since it is not used by this predicate manager.

Instead, it contains an array of Window Exploitation Managers, and each of these manages a WES. The applyPredicates() method behaves similarly to the way it does in Join Predicate Manager.

A.4 Exploitation Structures

PURPOSE: In Juggler, there are two Exploitation Structures. These are used to store tuples within a window size and also to quickly retrieve tuples to join.

Only two predicate types contain exploitation structures, join and window, Figure A.3. Each exploitation structure is used to retrieve tuples to join. The attribute predicate manager does not contain any exploitation structures and therefore does not produce any tuples to join. Only one predicate type, for any predicate type ordering, either join or window, can retrieve candidate tuples when processing an input tuple. For example, if a join predicate type is more selective than window, the join predicate manager's method applyPredicates() is called first. The join predicate manager will use its JESs to retrieve tuples to join since there are no candidate joined tuples from previous predicate type evaluation. When window predicate manager is processed, the applyPredicates() will be passed an Iterator of intermediate tuples. Since this was called after the join predicate manager, tuples were joined and the window predicate manager will only apply relevant predicates to these tuples.

If the window predicate type was to be more selective than join, the window predicate manager would use its WESs to retrieve tuples to join. In this case, the join predicate manager will only apply relevant predicates on the joined tuples that resulted from the previous evaluation phase. Therefore, the first predicate manager in the ordering of either join or window will use its corresponding exploitation structure to retrieve tuples to join. The predicate manager following it will only apply

predicates to these intermediate tuples and not use its exploitation structure.

If a query registered in the operator has at least one predicate for each of the three predicate types, the algorithm is used in its most optimal way. In the case where a query does not contain a predicate involving one of its streams, all the tuples in the exploitation structures for this stream will be returned in order to be joined.

A.4.1 Join Exploitation Structure

PURPOSE: The join exploitation structure is used to store and quickly retrieve tuples to join.

For each stream and column involved in a join predicate, a new JES is created. If there are 5 predicates, and none of the stream and column pairs are identical, 10 JESs will be created. This allows efficient retrieval of tuples using the input tuple's value. When an input tuple arrives into the operator, as a predicate is being evaluated, the tuple's column value will be used to retrieve tuples for the other stream to join. For instance, when evaluating the predicate $T.incr = H.vib$ and the input tuple T with values (1, 103, 101, 98), T's incr value of 98 is used to retrieve tuples from the JES structure for stream H. The tuples retrieved for candidate H are: {(0, 7, 98), (2, 7, 98), (3, 2, 98)}.

The array of JESs are contained in the Join Predicate Manager. The collection of JES's streams and columns represent all the join predicates' streams and columns registered in the operator. The maximum possible number of JESs in an operator is 2 * the number of join predicates registered in the operator. Since this algorithm assumes that most predicates registered in the operator are overlapping, the number of JESs in the operator will be less than the maximum. Without JESs to find relevant tuples to join, all of the tuples in the operator must be traversed.

Tuples in the JESs are removed when the tuples in a WES are identified as stale. Each tuple removed from a WES is also removed from any JESs that represent the stale tuple's stream. Duplicate insertions and wrongful deletions in the Red-Black trees are avoided by comparing timestamps. This also enables the operator to distinguish the stale tuple from another tuple with the same values.

There is room for optimization in the cleanup process. Currently, the JES stores copies of tuples. Java.util.TreeMap does not store references, therefore a copy of the tuple is used when inserted into a Red-Black tree. This means if there are three Red-Black trees for a stream, the tuple will be copied in each of them. During a WES cleanup, if stale tuples are set to null, the removal of these tuples from their relevant Red-Black trees is necessary. The JESs would be more efficient if an implementation similar to a Red-Black tree accepts and stores references. It also needs to recognize that if the reference is set to null, the tuple entry is also decremented and deleted. A possibility is using WeakRef instead of creating a copy of the tuple before it is inserted into a JES.

A.4.2 Window Exploitation Structure

PURPOSE: The window exploitation structure stores tuples in order of timestamp values and quickly retrieves tuples to join.

The number of WESs in an operator is determined by the number of input streams. If a query plan only contains one Juggler operator, the number of WESs in this operator is equal to the number of streams in this plan, or the number of input queues. If a query plan contains multiple Juggler operators, the number of WESs will be determined by the number of input queues of the operator.

WESs do not contain any structures provided by the Java API. A WES contains a collection of buckets and each bucket contains a JugglerWindowTupleList. This

structure will be described in a later section.

The number of buckets that each WES can contain is determined by the bucket limit value in the config file set during initialization time. The buckets will divide the timestamp range equally among themselves. If a tuple is outside the range of the last bucket, a new bucket is created. If the number of buckets exceeds the limit and the WES's time range has also exceeds the operator's maximum window size, the first bucket has become stale and all the tuples in this bucket are removed. While the tuples in the bucket are being removed, the copies of the tuples in the JESs are also removed. This process is described in Join Exploitation Structure section.

A.5 Predicate Structure

PURPOSE: The predicate structure is a representation of all the predicates registered in the operator. Predicates are normalized after initialization to allow comparing one predicate to another in order to identify similar predicates.

String representation of predicates cannot be used to order similar predicates. The Predicate structure allows ordering of predicates in a predicate lists. In config.xml, where predicates for each operator are specified, the predicates are in the format: stream1.column1 operator stream2.column2. During query and predicate registration, each of the predicate strings defined in the config file are parsed and converted to the Juggler's predicate structure. This structure is composed of two Juggler's PredicateParts and one Juggler operator, Figure A.4. In the case of an attribute predicate, there is a special predicate structure called AttrPredicate. This structure is composed of a Juggler PredicatePart, a Juggler operator, and a constant.

The Predicate architecture is based on Juggler's PredicatePart. This class is composed of a stream name, column name, and column index. The AttributePred-

icate is composed of only one PredicatePart since it compares to a constant, not another stream's value. Juggler Predicate representation for join or window predicates are composed of two PredicateParts. Both AttributePredicates and Predicates can be contained by several queries, and this is maintained in the predicate's list of queryIDs.

IPredicate is an interface which AttrPredicate and Predicate classes inherit. It requires the vital method getBit() be implemented. This method returns the predicate's BitPosition in the JugglerPBS and QED structures. This information links the predicate to Juggler's filtering structures.

A.5.1 Attribute Predicate

PURPOSE: Attribute predicates are a representation of the attribute predicates registered in the operator. It also allows comparisons of similar predicates after normalization.

Attribute predicates are defined in the config file in the format: stream.column operator value. While parsing, the stream and column are stored in the AttributePredicate's PredicatePart structure. Juggler's operator structure is defined in a later section. The value of the attribute predicate is stored in an AttributePredicatePart structure. The predicate's constant value is stored as a KeyInterface. This interface is specific to Juggler and it allows for many types of values. Currently, the type of values that have been tested and used is KeyDouble. It is a representation of a double number, but there is room for future enhancements to allow for more types of values and to compare different types interchangeably.

The operator is designed for limited streams and multiple queries with overlapping predicates. The predicate is listed once. To maintain the queries that contain a predicate, each predicate structure contains an array of queryIDs.

A.5.2 Join and Window Predicates

PURPOSE: Join and window predicates represent the join and window predicates registered in the operator. It also compares and orders similar predicates.

Join and Window predicates are defined in the config file in the same format: stream1.column1 operator stream2.column2. Both join and window predicates are represented by the Predicate structure. Since these two types of predicates contain two streams and two columns, the Predicate structure contains two PredicateParts. The operator is stored as a KeyInterface, which is similar to AttributePredicate's operator described above.

A.6 Predicate List

PURPOSE: Predicate Lists allow grouping of similar predicates and are also responsible for finding the most covering predicate for a tuple.

The collection of similar predicates are defined by the abstract class OrderList, Figure A.5. This class implements common methods for all predicate types. It contains a LinkedList of the predicates, the common operator among the predicates, the list index, and the predicate streams. The OrderList only contains one stream and column. The JWPredOrderList extends this class to also include another stream and column to accommodate the difference between attribute and join/window predicates. All lists require that all predicates it contains share streams, columns and operator. A predicate list is used to find the most covering predicate using a binary search, implemented by findMostCovering().

A.6.1 Attribute Predicate List

PURPOSE: The attribute predicate list orders attribute predicates in order of most to least covering.

AttrPredOrderList lists the attribute predicates in an ordered manner. This particular list contains only one stream and column, as defined in the abstract class, OrderList. The method, findMostCovering(), compares a tuple's column value, as indicated in the predicate, to a constant. This algorithm is dependent upon the predicate list's operator. For example, if the operator is =, the algorithm stops when the tuple's value is equal to the predicate's constant value. If the operator is >, the binary search on the list does not stop until the largest satisfying constant value is found.

The list's index indicates its position within the collection of attribute predicate lists. Using the list index and the predicate index in the list, the predicate's BitPosition correlates the BitSet in the Predicate BitSet Structure to the predicate.

A.7 Join/Window Predicate List

PURPOSE: Join and window predicate lists order join and window predicates in most to least covering. During tuple evaluation, the list is used to find the tuple's most covering predicates.

JWPredOrderList is an ordered list of join/window predicates. Join and Window predicate lists contain two common streams, two common columns, and a common operator. JWPredOrderList also extends OrderList. It implements findMostCovering(), but it differs in implementation from AttrPredOrderList. When an input tuple is processed over a join or window predicate list, the tuple's value is used to retrieve tuples from the join or window exploitation structure. If a candidate joined

tuple is processed over a join or window predicate list, the value of one stream and the value of the second stream are extracted from the joined tuple to determine if the predicate has been satisfied.

A.8 Predicate Lists

PURPOSE: Predicate Lists order predicates in most to least covering. Each list also contains a list index, which is used to correlate predicates to its position in the Predicate BitSet.

Predicate Lists play an important role in Juggler's algorithm. Finding the most covering predicate for a tuple is easier when using lists of ordered predicates. While the predicates are parsed and registered, the predicate lists are created. When a predicate is registered, the lists of predicates are probed to find a match. If a match is not found, a new predicate list is created. There are some differences in finding a matching predicate list between join/window and attribute. These differences will be described further in detail.

Attribute predicates are grouped in lists by AttrPredOrderListS and join/window predicates are grouped in lists by PredOrderListS. OrderListS is an abstract class which implements some of the common methods among the predicate lists. For example, getBitPosition() returns the predicate in the corresponding BitPostion parameter which describes the list index and position within the list. AttrPredOrderListS and PredOrderListS both implement the method findMostCovering(), which differs in its implementation between predicate types.

A.8.1 Attribute Predicate Lists

PURPOSE: Attribute Predicate Lists contain predicate lists of attribute predicates. It is used by the attribute predicate manager and aids in finding a tuple's most covering predicate.

AttrPredOrderListS extends OrderListS and implements methods specific for attribute predicates. For example, findMostCovering() for attribute predicates lists is different from both join and window's method, but due to the difference in the predicate structures, have similar logic in their findMostCovering() method. If an input tuple is processed, the predicate lists, if relevant to the tuple, are processed to find the most covering predicate in each. If an intermediate tuple is processed, only the predicates that are indicated as relevant by its RelPBS are processed.

A.8.2 Join/Window Predicate Lists

PURPOSE: Join and window predicate lists order join and window predicates respectively. It is used to find the tuple's most covering predicates.

PredOrderListS contains a collection of join or window predicate lists. PredOrderListS is a collection of only join predicate lists, or JWPredOrderLists. PredOrderListS are similar to AttrPredOrderListS but differ in implementation of the method findMostCovering(). This method traverses through each predicate list to find the most covering predicate.

A.8.3 BitPosition

PURPOSE: The BitPosition structure aids in correlating the predicates in the predicate lists to the PredicateBitSet structures.

BitPosition contains the predicate list's index and the index of the predicate in

that list. These two coordinates indicate the predicate's bit position in the PredicateBitSet structure, JugglerPBS, RelPBS, and query QED. All predicate types use this structure since all predicate type managers contain predicate lists.

A.9 BitSet Collection

PURPOSE: This is an array of BitSets, representing predicates in a Predicate List.

Predicates are grouped into lists that have similar streams, columns, and operators. Predicate Lists are contained by predicate types. Each query is represented by the group of all three predicate types and their predicate lists.

Each predicate in the predicate list is represented by a bit in a BitSet. All predicates in a predicate list are represented by an array of BitSets, Figure A.6. Each predicate type has a collection of all its predicate lists. The collection of all three predicate types are used to represent a query or a tuple's filtering predicate structures, JugglerPBS, SatPBS and RelPBS.

A.9.1 General Predicate BitSet

PURPOSE: General Predicate BitSet represents Predicate Lists as collection of BitSets.

This structure represents predicate lists as an array of BitSets. It also contains the name of the predicate type that the list pertains to. Methods that compare if BitSets are supersets of each other are used to process the tuple's relevancy to queries. Another method, setMostCoveringBits(), sets the index returned from the predicate list's findMostCovering(), and all other indexes that have also been currently satisfied.

A.9.2 Predicate BitSet

PURPOSE: Predicate BitSet represents Predicate Lists of a predicate type as bits.

PredicateBitSet extends GeneralPredicateBitSet. This represents one predicate type for a query or for a tuple. This class implements IPredicate, which requires the implementation of methods such as isSuperSet(). This methods is used to evaluate a tuple's relevancy to a query's QED. This is vital to the Juggler's join algorithm.

A.9.3 ExtGeneralType BitSet

PURPOSE: ExtGeneralType BitSet represents tuples for one predicate type.

The ExtGeneralTypeBitSet is composed of two GeneralPredicateBitSet for a predicate type. One of these GeneralPredicateBitSets represents the relevant predicates in a list while the other represents the satisfied predicates. Therefore, to represent each predicate type, three ExtGeneralTypeBitSet are needed, one for each predicate type.

A.9.4 ExtPredicate BitSet

PURPOSE: ExtPredicate BitSet represents tuples JugglerPBS for all predicate types.

This represents all of the predicate types in the Juggler operator. It contains an array of ExtGeneralTypeBitSets. Since we have defined only three predicate types, there will be three ExtGeneralTypeBitSets. This represents a query consisting of all its predicate types, each of the predicate type's lists, and the predicates in each list. It is also used to represent a tuple's filtering scorecard. JugglerPBS, an ExtGeneralTypeBitSet, is composed of RelPBS and SatPBS. This is a crucial structure to Juggler's join algorithm processing to validate a tuple's relevancy to

queries.

A.10 Juggler Predicate BitSet Structure

PURPOSE: JugglerPBS is used by tuples to maintain its predicate evaluation history.

JugglerPBS, Juggler Predicate BitSet Structure, is a filtering structure for every tuple. As a tuple enters into a Juggler operator, JugglerPBS is attached to the input tuple. JugglerPBS is composed of RelPBS (Relevant Predicate BitSet Structure) and SatPBS (Satisfied Predicate BitSet Structure). Both of these structures are logical structures that aid in calculating the tuple's relevant predicates and the tuple's candidate queries.

A.10.1 Relevant Predicate BitSet Structure

PURPOSE: RelPBS identifies unnecessary predicates that do not need to be evaluated.

RelPBS is the logical structure of ExtGeneralTypeBitSet's ExtPredicateBitSet. This contains relevant GeneralPredicateBitSets for all three predicate types. It is only a way to group the predicate type's relevant BitSet collection, GeneralPredicateBitSets. During tuple processing, RelPBS indicates which predicates need to be evaluated for the tuple reducing the number of comparisons. It is also used to quickly assess if a tuple will satisfy a query registered in the operator with simple BitSet comparisons.

A.10.2 Satisfied Predicate BitSet Structure

PURPOSE: SatPBS maintains a tuple's evaluated predicates.

SatPBS, Satisfied Predicate BitSet, is also a logical structure for all three predicate types, similar to RelPBS. This structure represents the predicates that the tuple has satisfied. Comparing the tuple's SatPBS, RelPBS and QED, a tuple's candidate query and relevant predicates are quickly identified. BitSet manipulation provides a quick and easy mechanism of filtering intermediate tuples early in the join algorithm, thereby reducing the number of intermediate joined tuples in the operator.

A.11 Query Encoding Dependency

PURPOSE: QED represents queries and its bit representation of its Predicate Lists.

QED, Query Encoding Dependency, is a structure that aids in filtering candidate tuples in the join process. A QED is represented by a ExtPredicateBitSet which is a collection of predicate lists for all three predicate types. The satisfied BitSet representation of ExtPredicateBitSet is not used. This structure only represents all the predicates that the query requires. Only one of the GeneralPredicateBitSet is used in this structure. Every query registered in a Juggler operator has a QED associated.

A.11.1 Juggler Operator's QEDs

PURPOSE: QED is a representation of a registered query.

The Juggler operator has multiple queries registered. The collection of QEDs for all the registered queries is called JugglerQEDs. The size of the QED array is determined by the number of queries registered in the operator. This structure is used to find all candidate queries for a tuple by comparing each of the query's QED to the tuple's JugglerPBS. The query QED is compared to the tuple's RelPBS. The

BitSets that are indicated as relevant to the tuple and required by the query are then checked to see if they have been satisfied by comparing the tuple's SatPBS. If all the common BitSets in RelPBS and the query's QED are satisfied in the tuple's SatPBS, the query is a candidate tuple for this query. To assess if a candidate tuple has been satisfied by a query, the query's QED and tuple's JugglerPBSare compared again. If the common BitSets include all the BitSets in the query's QED, and all the common BitSets have been satisfied in the tuple's SatPBS, the tuple has satisfied the query.

A.12 Juggler Tuple

PURPOSE: The JugglerTuple stores tuple values, a JugglerPBS, and a candidate query list.

JugglerTuple extends XATTuple. When a XATTuple is dequeued from the queue and enters the Juggler operator, it is cast as a JugglerTuple. This structure adds some structures to XATTuple that is specific for the Juggler operator's algorithm. Every JugglerTuple has a JugglerPBS, or ExtPredicateBitset, which maintains the relevant and satisfied predicates for the input or candidate joined tuple. The JugglerTuple also maintains a list of candidate queries. These queries are updated after a predicate type has been evaluated. Tuple's XATTimestamp's MaxTimeStamp is used to evaluate window predicates.

A.12.1 JugglerTupleList

PURPOSE: JugglerTupleList is a collection of Juggler Tuples before joining.

JugglerTupleList is a linked list of JugglerTuples. This list holds the each tuple in the intermediate joined tuple until the tuple is finally joined before it is output.

It also enables the tuples to maintain their own JugglerPBS. For instance after an input tuple has been processed, it is stored in the operator's exploitation structures. The input tuple's attribute predicates that have been evaluated are stored along with the tuple. This avoids repeated predicate evaluation when this tuple is joined with other new input tuples. When tuples are retrieved from the exploitation structures, the retrieved tuples will already have their satisfied attribute predicates set. The retrieved tuple's JugglerPBS and the input tuple's JugglerPBS can be combined before joining and comparing the joined tuple's relevancy to queries in the operator.

JugglerTupleListIterator extends the Iterator and allows for easy traversal over the collection of JugglerTuples. This is a convenient class, which only simplifies implementation.

A.12.2 JugglerWindowTupleList

PURPOSE: JugglerWindowTupleLists are a collection of JugglerTupleLists used by WES and JES.

A collection of JugglerTupleLists is a JugglerWindowTupleList. This structure is used while an input tuple is being processed, resulting in multiple candidate joined tuples or JugglerTupleLists. This structure avoids adding duplicate JugglerTupleLists in the JugglerWindowTupleList. During join or window predicate type processing, each relevant predicate is used to retrieve candidate tuples to join. If during this process duplicate tuples are returned, they are not added to this structure.

Also, WES uses this structure to store tuples in its buckets. Each bucket is created for each of its input queues. This structure allows the ability to store JugglerTupleLists.

JugglerWindowTupleListIterator extends Iterator and allows for easy traversal

over the collection of JugglerTupleLists. This is another convenient class which only simplifies implementation.

A.13 Juggler Comparative Keys

PURPOSE: Juggler's comparative keys are used in comparing tuple values. It is also extendable to allow for future extensions.

KeyInterface defines the methods required for implementing all its inheriting classes: KeyString, KeyLong, KeyInt, and KeyDouble. The structure allows for future implementation of comparing one type to another, for example, KeyString to KeyLong. It can allow for values to be compared, such as urls, request objects, or server addresses. The Juggler operator currently evaluates predicates using KeyDouble. Every tuple value involved in a predicate is cast as a KeyDouble during evaluation.

A.14 Juggler Comparative Operators

PURPOSE: Juggler's comparative operators are used in comparing similar predicates and tuple evaluation.

The operator interface defines the methods required to be implemented by all Juggler comparative operators: EqualToOperator, LessThanEqualToOp, LessThanOp, GreaterThanEqualToOp, and GreaterThanOp. When the predicates are parsed, the predicate operators are parsed into an Operator type. Each of the Operator's inherited classes must implement evaluate(). These methods return true if the value satisfies the Operator. These classes are used throughout predicate evaluation. All classes have been tested and are used in the Juggler Operator.

Figure A.1: Juggler Operator Interface

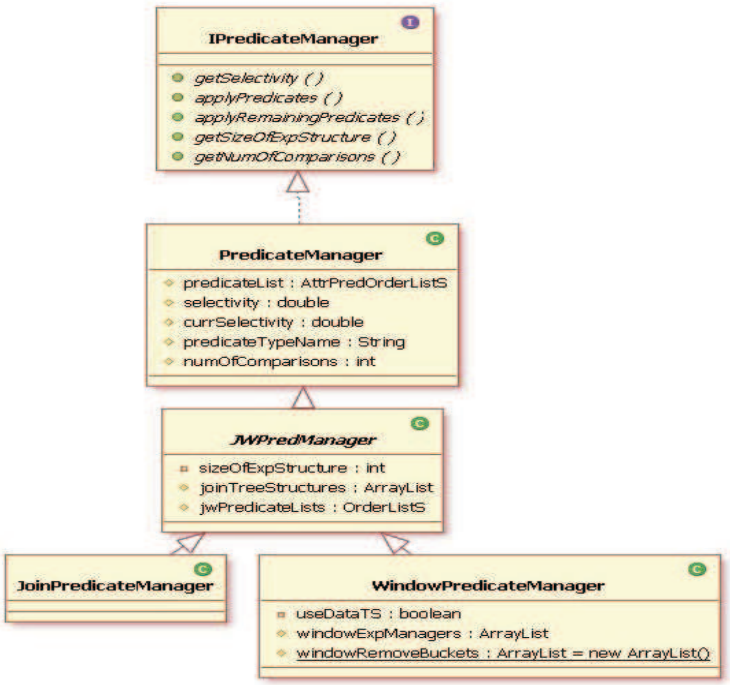

Figure A.2: Predicate Type Managers

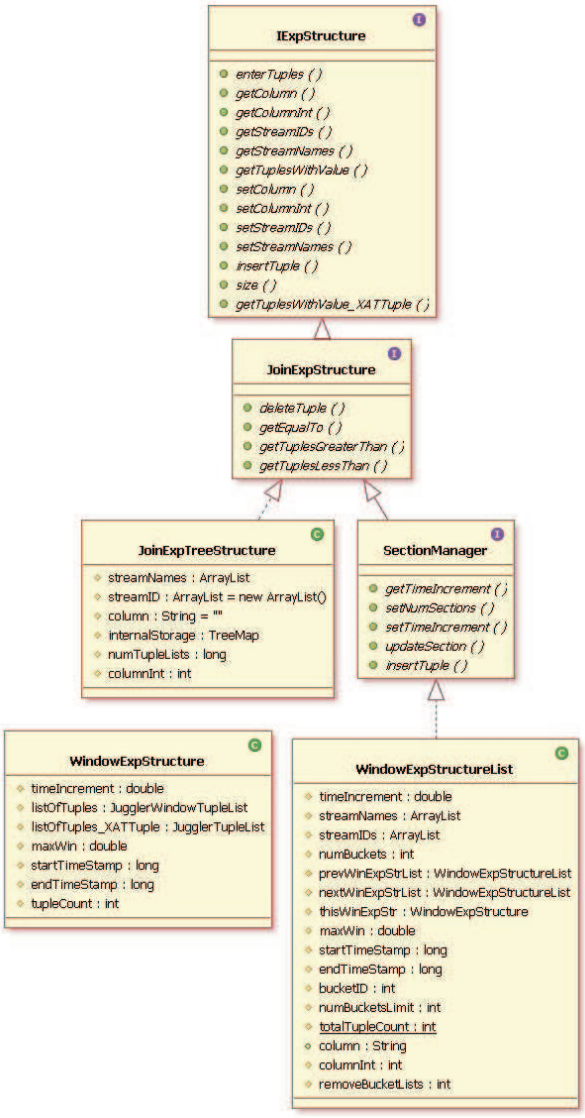

Figure A.3: Predicate Exploitation Structures

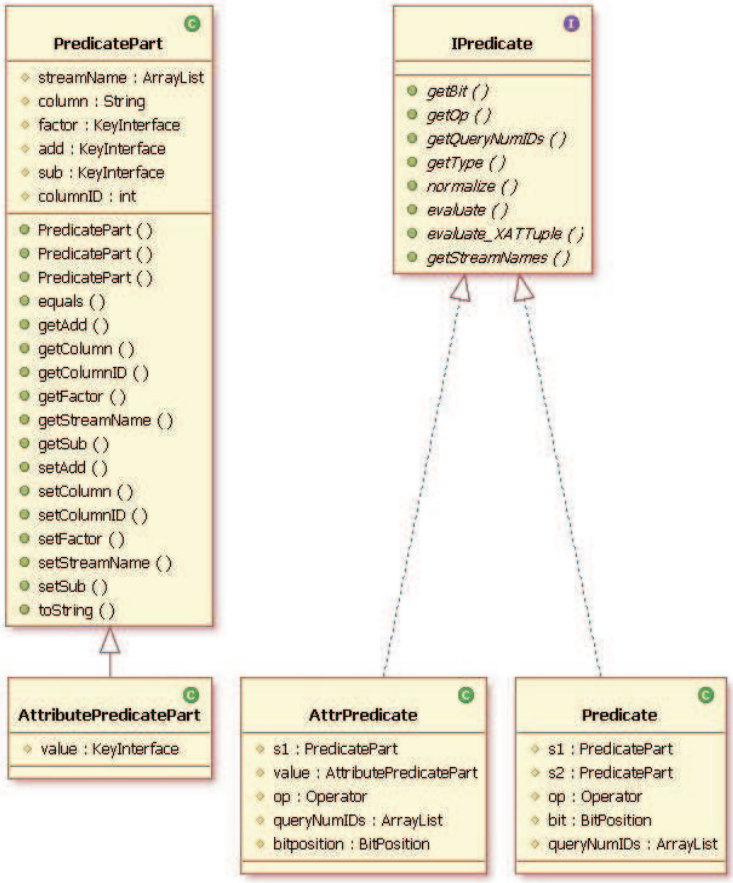

Figure A.4: Predicates

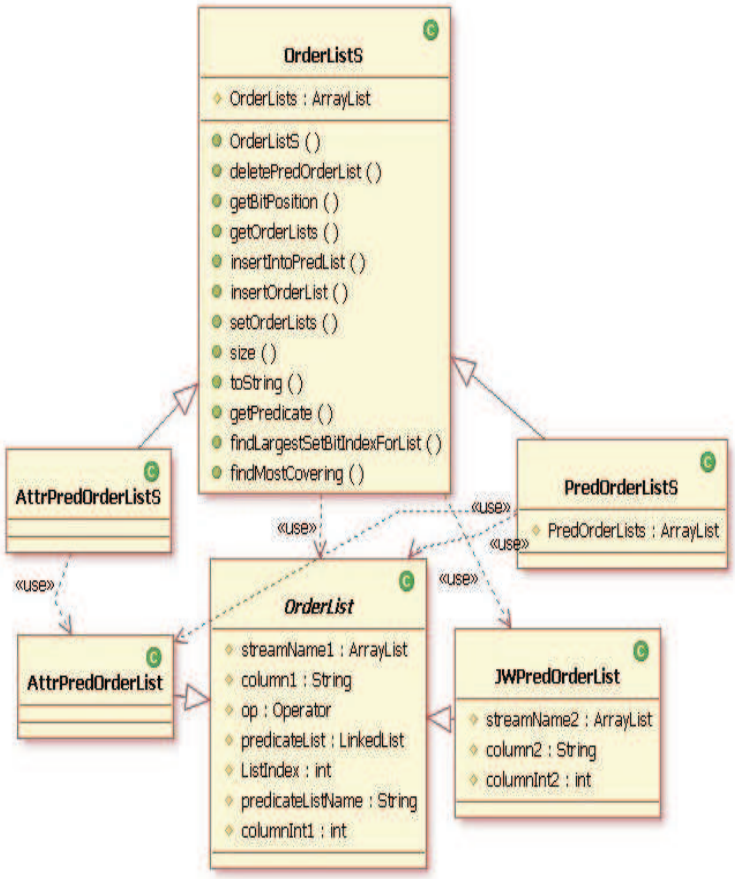

Figure A.5: Predicate List Structures

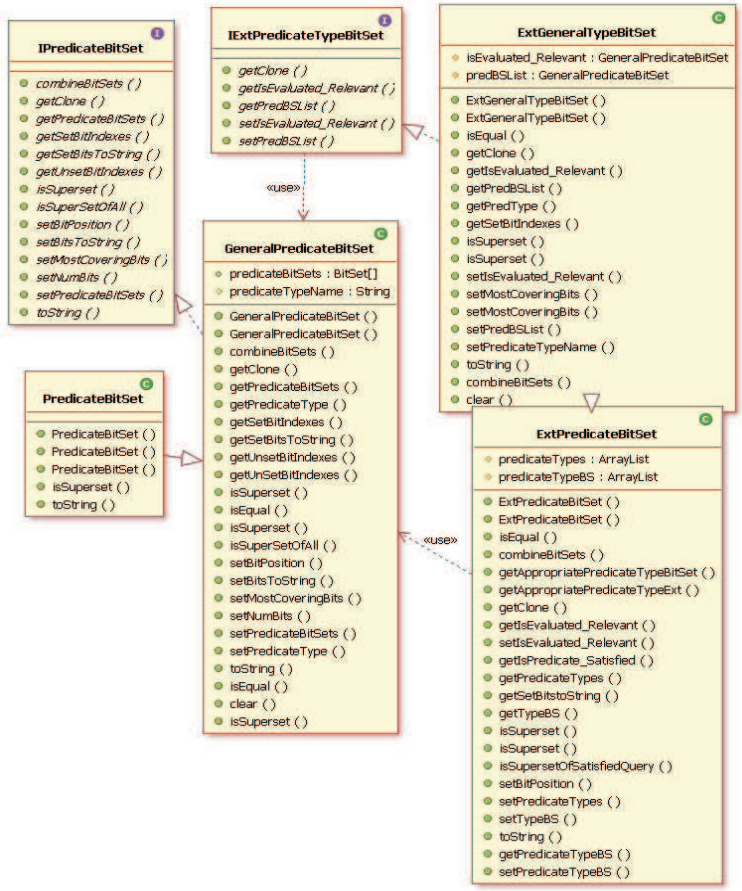

Figure A.6: Predicate BitSet Structures

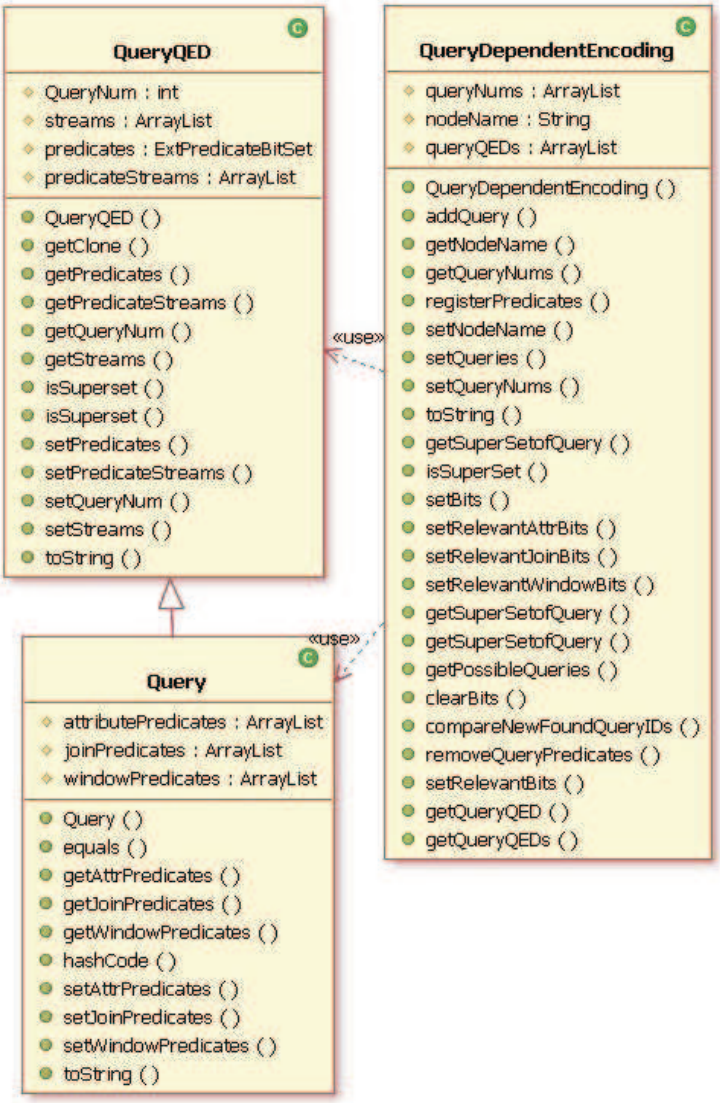

Figure A.7: Query Structures

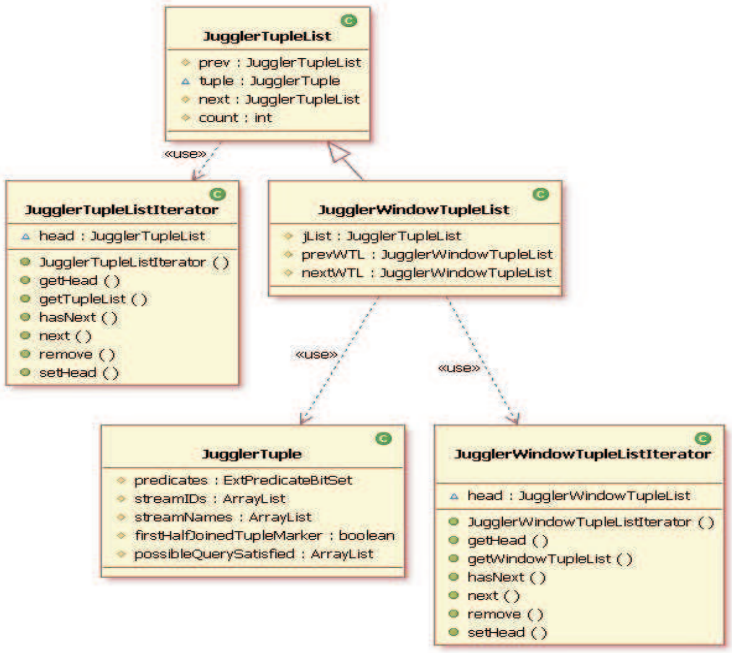

Figure A.8: Juggler Tuple Structures

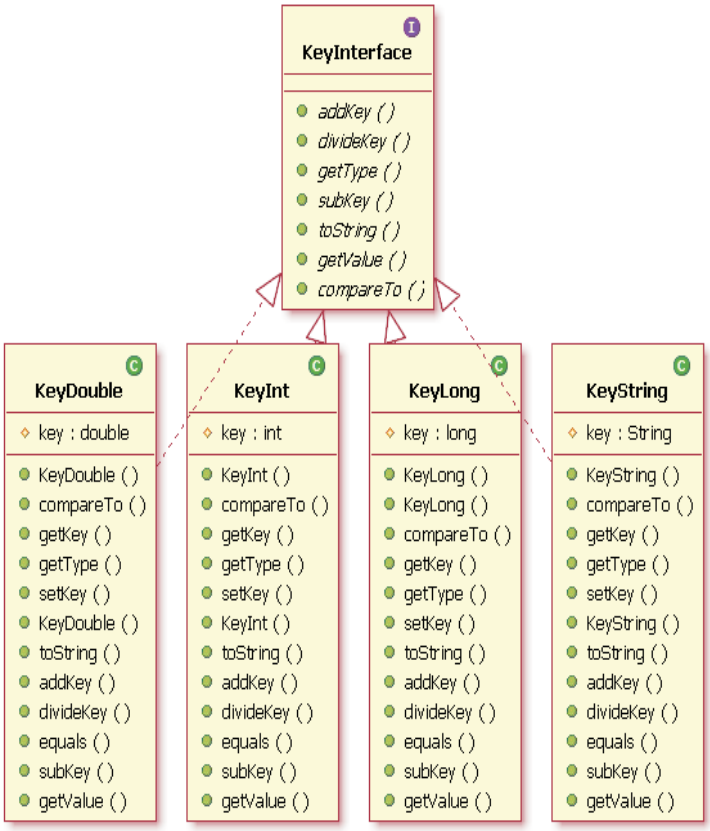

Figure A.9: Juggler Comparative Keys

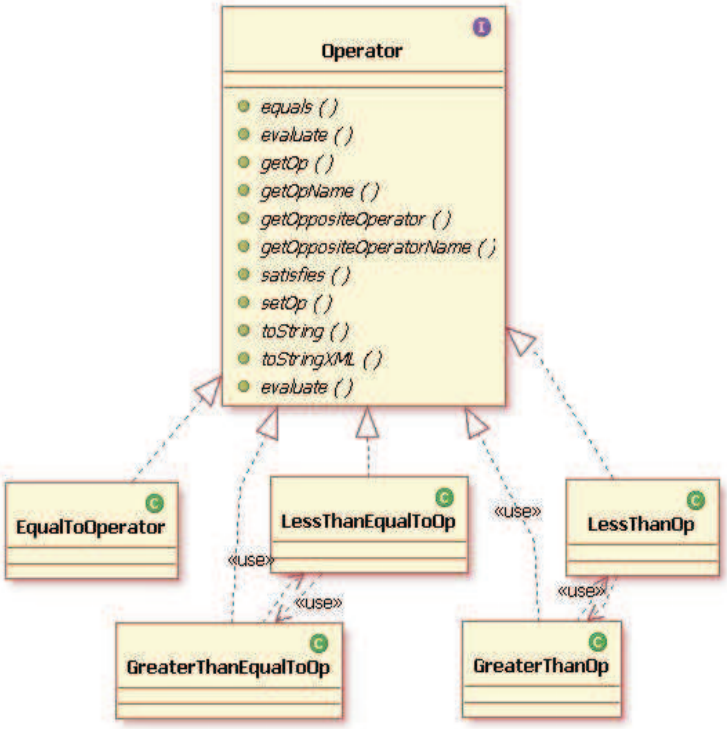

Figure A.10: Juggler Comparative Operators