# Multi-Mode Stream Processing For Hopping Window Queries

by Mingrui Wei

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

---

May 2008

APPROVED:

---

Professor Elke Rundensteiner, Thesis Advisor

---

Professor George T. Heineman, Thesis Reader

---

Professor Michael A. Gennert, Head of Department

**Abstract**

Window constraints are mechanisms to bound the tuples processed by continuous queries specified over unbounded data streams. While sliding window queries move the constraint window upon the arrival of each individual tuple, hopping window queries instead move the window by a fixed amount after some period, thus periodically refreshing their results. We observe that for large hops, techniques liked delta result updating may not be efficient – as large portions of the tuples in the current window will be different from the previous window and thus must be maintained. On the other hand, the complete result updating technique, which has been found to be less suitable for sliding windows queries. Compute the next result based on the complete current window now can be shown to be superior in performance for some hopping windows queries. A trade-off emerges between the complete result method which has a lower per tuple processes cost but potentially processing redundant results versus the delta result method which has no redundant processing but pays a higher per tuple processing cost. On top of that, strict non-monotonic operators such as difference operator, cause premature expiration due to operator semantics. Negative tuples are needed for this kind of special expiration. Such negative tuples added extra burden to the stream engine. Thus, in streaming processing, the difference operator is typically suggested to be placed on top of the query plan despite its potential ability to reduce cardinality of the stream. With this thesis, we introduce a whole solution for hopping window query processing which includes an optimizer for generalized hopping window query optimization that exploits both processing techniques within one integrated query plan alone with query plan rewriting. First, we design the query operators to be multi-mode, that is, to be able to

take either a delta or a complete result as input, and produce either a delta result or complete result as output. Then we design a cost model to be able to chose the optimal mode for each operator. Thirdly, our optimizer targets to configure each operator within a query plan to work in the suitable mode to achieve minimum overall processing costs. Last but not least, two query optimization techniques have been adopted. One explores all possibilities of pushing the difference down past joins using dynamic programming and assigning optimal mode at the same time, the other applies heuristic difference push down rule. The proposed techniques has been implemented within the WPI stream query engine, called CAPE. Finally, we show the benefit of our solution with a vast number of experimental results.

**Acknowledgements**

I would like to express my gratitude to Luping Ding, Abhishek Mukherji, Song Wang, Venky Raghavan and all other DSRG members for being supportive. My thanks also goes to Prof. Heineman for his support. Most Importantly to my advisor Prof. Rundensteiner for her incredible patience, guidance and support that guided me every step alone the way.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background on Stream Processing

Stream data processing has grown into a popular research field due to the diversity of modern applications requiring stream processing such as sensor networks [19] [25], real time event analysis [18], traffic analysis [20] and web-based financial tickers. These applications required monitoring capabilities from the data processing engine. During such monitoring activities, humans are usually passive and the Data Stream Management System (DSMS) is active [2]. In such environments many assumptions made in traditional query processing are no longer valid, and clearly new unstudied issues arise.

One of the fundamental issues that arises is how to process unbounded data in real time. The most practical method currently employed is to introduce "window predicates" that restrict the number of tuples that must be processed for each stream at a time. There are two basic approaches: time-based windows and count-based windows [11]. The time-based window is defined using some notion of a clock

that ticks independently of the number of tuples received during the time period. Tuples that arrive between the start and the end of ticks are said to fall within that window. Thus, each tuple that belongs to a particular window has a timestamp within the start and the end time of the window. The actual number of tuples in that window varies from window to window. Such number is determined in large by the distribution of the arrival rate. The count-based window is defined based on the number of tuples. In contrast to time-based windows, the number of tuples in each count-based window is fixed. In order to keep the discussion simple, we henceforth focus on time-based windows. However our approach holds true for count-based windows as well. Because count-based windows can be seen as time-based windows with a uniform distribution.

Another issue to be determined is how to move the window over the input stream when queries are running continuously. Currently, three ways have been used to move the window from one direction to another over the input stream. The first called sliding windows [14] [5], where the window size remains fixed and both the upper and lower window boundaries move smoothly in unison one element or one time unit at-a-time. The second is called landmark windows [9], where the starting point of the window stays fixed while the other end moves forward. This will make the window size increase as time passes. The final one is called hopping or tumbling windows. The window size remains fixed but the window boundaries move in a discontinuous fashion. Several works [17], [16], [7] have discussed these different window semantics. However the majority of systems [1] [3] deal with continuous queries using sliding windows only, that is they use the time stamp of each newly arriving tuple as the end of the current window and refresh the answer each time a new tuple arrives.

## 1.2 Motivating Example

A commander may want to know where are his soldiers and how are they doing. But he does not want to be bothered too often. He only wants such information to display on his monitoring screen every half minutes. An example of such hopping query specified in a CQL [4] liked language is :

(SELECT GPS, soldierID RANGE 1 minute
FROM Sgps, Shealthy
WHERE Sgps.ID=Shealthy.ID
)
MINUS (SELECT GPS, soldierID RANGE 1 minute
FROM Sgps, Ssupplylow
WHERE Sgps.ID=Ssupplylow.ID
)
Hop by 0.5 minutes

The above query produces an answer set that contains the soldier ID and GPS position of healthy soldiers that are low on supply. Shealthy, Ssupplylow and Sgps are sensor readings. The query is different from its sliding counterpart. While the sliding query outputs whenever a new tuple arrives, the hopping query will output its result every half minute whenever the window hops. As mentioned above, the hopping window semantics are useful for many applications especially for monitoring purposes. This setup would also have has less pressure on the system resources as

it only computes answer every 0.5 minutes instead of for every new tuple.
In this thesis, we instead focus on hopping windows for the following reasons:

- Physical limitations of machines and humans. The ability to display or notify the end user. For example, if we use a TV monitor as our display, only 24 frames can be displayed in a second. This makes it unnecessary to update the results any faster than what they can be consumed by display devices. Besides that, human's physical limitation also has to be taken into consideration too. Humans usually react to information at the granularity of seconds. However, computers react and process things at the granularity of millisecond. Humans will be overwhelmed by the volume of data produced by machines.

- Accumulate changes. Often times, the changes per every tuple are very small, sometimes not noticeable. On the other hand, changes between hops are easier to discern.

- User intention [16]. A user may want to only be notified periodically.

## 1.3 State-of-Art Review

In window-based query processing the question of how to efficiently present the results for each window is important. The presentation should contain enough information to indicate changes between windows. Two alternative methods could be applied to indicate such changes, namely to explicitly or implicitly show the changes. Research efforts [2] [3] have been conducted to support both indication in data stream management systems, called delta result (explicitly) and complete result (implicitly). The former presents the changes explicitly by showing the delta between two adjacent answer sets. On the other hand, the later presents the current

valid results regardless of changes. Two alternative methods aimed at most effectively supporting these two methodology for producing answers. These two methods are query reevaluation and query incremental evaluation methods.

Existing techniques select one of these methods on a given query as a whole. Incremental evaluation usually requires delta result input because delta result indicates only the changes by which the current window differs form the previous window. As the window slides, the changes in the window can be represented by the two sets of inserted and expired tuples respectively. An incremental style query operator is used in the query to process both the inserted and expired tuples and to produce the incremental changes to the query answer as another set of inserted and expired tuples.

Two approaches have been adopted in the literature to support Delta Result Approach(DRA for short), namely, the input-triggered approach (ITA) and the negative tuple approach (NTA)[10]. NTA is introduced as the delay-based (to minimize delay) uniform framework (to handle mature expired and premature expired tuples the same way) that will produce positive tuples for the insertion set and negative tuples for expiration set explicitly. This approach has the major drawback that the system has to process double the amount of tuples because eventually every tuple will expire (except for group-by operators which can stop negative tuples from propagating [12]). On the other hand, in ITA, only the newly inserted tuples flow through the query pipeline. The operators in the pipeline rely on the timestamps of the inserted tuples to expire old tuples. However negative tuples will still be needed for invalidating (premature expiration) tuples because the invalidated time is not determined by timestamps but by operator semantics, for example for set difference operators. Operators that must send out negative tuples to invalidating

its own result is called strict non-monotonic operator [12]. ITA can also suffer from significant delays if no tuples arrive for a long time.

The reevaluation method on the other hand requires the Complete Result Approach (CRA for short). CRA does not explicitly indicate changes, which means there are no sets of insertion or expiration tuples. Changes made to the previous window which the next operator needs are implicitly indicated by the current complete result instead of produce a insertion set and expiration set. When using CRA, a new valid result will be generated for each window regardless of whether the changes occurred. It also does not need to process negative tuples. However, the disadvantage of CRA is that repeated work will have to be performed often if the outputs of two adjacent windows are very similar.

In the sliding window semantics, since the output is updated at the arrival of each new tuple, a large portion of the current result will be similar to the previous one. Thus only a small amount of positive tuples and negative tuples, if any, need to be sent out when using the delta method for sliding window semantics. The cost of reevaluating the whole window is much greater than sending out only the changes. Thus the delta result updating method has become a natural standard for implementing the sliding window semantics in the literature.

However, we observe that for hopping window semantics, many tuples that belong to the result from the previous window may not still be valid for the current window. For example, at the 50% Hop/Window ratio, on average half of the tuples in the current window will no longer belong to the next window. Hence a larger amount of the results may have to be changed compared to slide by one tuple case.

One needs to potentially both expire a large number of tuples belonging to the previous result and to add a large number of newly produced results. In such case, CRA is preferred because there is no need to maintain states and little computation is redundant.

## 1.4 Approach

As we stated above, both the delta result and the complete result approaches have their advantages and limitations. We propose to combine these two update methods into a hybrid solution to minimize the cost of executing hopping window query plans. First, we redesign the standard relational operators so that each of them has multi input and output modes. Therefore they can minimize output tuples and system costs by running the more suitable input and output combination. Namely, the input mode can be either a delta result or a complete result. Similarly output mode can also be either a delta result or a complete result.

In our hybrid query plan, each operator can be configured to work in any input and output mode combination independent of its predecessor or successor nodes. Such configuration is done according to our cost model, henceforth also called mode assignment. Which mode to assign to an operator will depend on factors like the size of the hop, the window size and the distribution of the data and the cost of the query plan. Our approach will provide the optimizer the most flexibility to push down the operators having the lower selectivity independent of whether or not they produce negative tuples. Because we can convert a delta result to a complete result, we can hide the existence of negative tuples if desired.

Contributions of this thesis include:

- We study the trade-off of the DRA and CRA for hopping window semantics. We propose to incorporate these two in a single the hopping window query plan as a whole to achieve a maximum throught put.

- We investigate the behavior of the class of strict non-monotonic operators (for example, difference and antijoin). We also design an efficient physical design for this class of operators and rewrite rules that indicate when it is semantically correct to reposition operators belong to this class in a query plan.

- We redesign the set of core stream relational algebra operators to have them equipped with dual input and output modes of processing as well as suitable data structures and algorithms.

- We design a mode assigner that configures each operator within a given query plan to run in the best input and output mode combination. We have proven our mode assignment produce optimal mode assignment for the query plan. It is also shown to have a linear complexity in the number of operators in the given query plan.

- We extend the conventional cost-based optimizer with rewriting rules and plan search algorithms that consider the operator mode assignment problem and operator positioning problem. So Join, Select, Project, Difference, Antijoin will all be considered during optimization without sacrificing asymptotic time complexity.

- We develop a heuristic to reposition a strict non-monotonic operator in a query plan to achieve maximum through put.

In the remainder of this thesis, Chapter 2 describes the preliminary material. Chapter 3 contains the operator semantics. Chapter 4 describes the execution strat-

egy. Chapter 5 presents the physical implementation of each operator and the cost model according to the implementation. Chapter 6 discusses the query plan optimization and rewriting heuristic rules. Chapter 7 shows experimental results. Chapter 8 consists of related work. Chapter 9 concludes the paper and suggests future work.

# Chapter 2

# Preliminaries

## 2.1 Operators

In our system, we work with the core relational algebra composed of selection, projection, cartesian product, set union and set difference. We chose to deal with these operators because they are fundamental in the sense that none of them can be omitted without losing expressive power [8]. Many other operators have been defined in terms of them. In addition to these operators, group-By, and anti-join are also supported in our system. More details of their semantics are given in Chapter 3

## 2.2 Query Execution Strategies

In data processing, in general we can process data after a batch of them has accumulated. Or we can also eagerly process data as they arrive, doing a little bit of work each time. These two alternative processing approaches have been used in many different contexts, but rarely have been investigated carefully in the streaming processing context. Typically, most research will either pick one or the other method of query evaluation. To our best knowledge, the comparison and study of

the usage of these two strategies has never been conducted for hopping queries. We will discuss briefly these two strategies in this section and further explain them in more detail in Chapter 4.

In hopping processing, if the operator is stateless (does not require to access other tuples in the same window), it processes its input on the fly. It does not have the notion of gathering information since it does not need information about other tuples in the same window. Thus it does not matter what strategy it uses. However, if it is stateful, such strategies will make a difference. If a stateful operator gathers tuples during each window and maintain the data in its states as necessary. It only evaluates its state tuples when the current window is complete and no more tuples belonging to the current window will arrive. Since it applies its semantic to all the tuples in the current window at once, we refer to this strategy as window-at-a-time (WAAT) .

On the other hand, the operator can process its input tuples more eagerly and apply its operator semantics to each tuple as they arrive, without accumulating the full content of the current window. Since it processes tuples incrementally, we will refer to this strategy as the incremental method or tuple-at-a-time ( TAAT ) method.

We briefly compare the advantages and disadvantages of these two execution strategies.

- WAAT execution: This method is (particularly) a blocking strategy in the sense that the next operator will not be able to run until the previous operator has finished all its work. The output of an operator using this strategy is likely to be fluctuating because result tuples will be produced at the end of each window. This may cause significant delay if the hop is huge. The advantage of the WAAT strategy is that it has all the tuples in the current window, thus

any potentially unnecessary processing can be avoided.

- TAAT execution. This approach has been widely use in stream processing system for sliding window queries. Often, this is non-blocking, that is the next operator will be able to see the partial output immediately and start processing without the previous operator having finished its job for the current window content. It is likely to get a more steady output stream as output may be produced quickly as tuples arrive. However, when the output is being produced without seeing all the tuples in the current window, the output may not be 100% certain in some cases. Mode detainl in chapter 4. When this arises, then higher overall processing cost may occure due to having to correct its output over time until finaly the whole window has been seens.

## 2.3 Hopping-Window Query Semantics

A hopping-window query is a continuous query over n input data streams, $S_1$ to $S_n$. Each input data stream $S_j$ is assigned a window of size $w_j$ and a hop of size h. While different streams may have different window sizes, where the hop size is fix for the whole query in order to synchronize the processing of different operators. This way all windows are moving forward at the same speed. The answer to the hopping-window query is equal to the answer of the snapshot query whose inputs are the elements in the current window for each input stream for each hop.

# Chapter 3

# Operator Semantics

Two issues should be distinguished when discussing hopping window operators: operator semantics and operator implementation. Operator semantics defines how the changes in the operator's input will affect the operator's output and how the outside world sees the operator. On the other hand, operator implementation defines the actions the operator takes when the input changes to achieve the desired semantics. It depends on whether delta or complete input is used and whether delta or complete output is needed. In this chapter, we first discuss the semantics, then analyze issues for the various relation operators under all four input output combinations, namely Delta In, Complete Out (DICO for short), Delta In, Delta Out (DIDO for short), Complete In, Complete Out (CICO for short) and Complete In, Delta Out (CIDO for short). Implementation is describe in section 5.1 for each of them.

## 3.1 Auxiliary Operator

**Definition 1** (Input Output Format Conversion)**.** *For an operator to perform a conversion is to say, the operator consumes its input in either complete or delta format and produces its output in a form that is different from its input format.*

In order to simplify the discussion, we assume that there are auxiliary operators in every source stream to convert the source stream to either a delta or a complete stream which is comply with the window size and hop distance. By convention, we assume that there are no negative tuples in the source streams.

## 3.2 Window Select and Window Project

Select is an unary operator that only outputs tuples that satisfy a condition p. It is sometimes called filter operator or restriction operator. Project is also an unary operator that only outputs tuples that are restricted to the set of attributes in A. Conventionally, in streaming data processing system both of them do not care about the sign of the input tuples; both positive and negative tuples are processed the same way. And both select and project will not change the tuples' timestamp. However, incorporating hopping semantics and the notion of input and output conversion, the semantics has also changed. If the input mode is equal to the output mode, select and project work like the same operators in the sliding window queries. If the input mode is not equal to the output mode, select and project operators will perform a conversion. In the DICO case, it will consume its delta input and produce a complete output. In the CIDO case, it will consume its complete input and produce a delta output.

## 3.3 Window Join

Natural join is a binary operator. The output tuples are the concatenation of tuples from the input stream S and R that satisfy the join condition. If we let the join condition to allow all tuples to concatenate, window join is equal to cartesian product. Unlike select and project, timestamps of the output tuples produced by join

are different from its input. It consists of two parts, a maximum timestamp equal to the maximum value of the timestamps from the joined tuples. A tuple also has a minimum timestamp (some time it is called expiration timestamp) that equals the minimum value of timestamps from the joined tuples.

Since join is symmetric, we discuss the effect of negative and positive tuples for only one side of the join. In DIDO case, the join operator consumes its delta input and produces delta output for the current window and updates its result per hop. Both the negative and positive tuples are processed the same way. It is the same as the sliding window join operator processing its input. In DICO case, the join operator consumes its delta input, performs conversion and produces complete output for the current window and refresh it complete result per hop. Such conversion is done by applying the delta result produced by the current window to the previous output cache which keeps the complete previous result. In CICO case, the join operator consume its complete input and produce a complete output for the current window and refresh its result per hop. Since there are no negative tuples, the operator processes the input the same as the sliding window join. In CIDO case, the join operator consumes its complete input, performs conversion, and produces delta output for the current window and refreshes its result per hop. Such delta is determine by comparing the current output cache with previous output cache. Tuples in the current output cache but not in the previous output cache will be outputted as new positive tuples. Tuples in the previous output cache but not in the current output cache will be outputted as negative tuples.

## 3.4 Window Set Operators

We support set union and set difference in our stream processing engine. For set union and set difference, the two relations involved must be union-compatible. That is, the two relations must have the same set of attributes. As set intersection can be defined in terms of set difference, the two relations involved in set intersection must also be union-compatible.

Union operator outputs tuples of stream S and R. It simply combines them together to make one stream in the order of timestamps of tuples.

### 3.4.1 Window Union

Union is a binary operator. An input tuple to the union operator is produced in the output with the same sign. Unlike join, an output tuple carries the same timestamp and expiration timestamp as the input tuple. For both the CICO and DIDO cases, their is no extra work to be done. Union will process both the positive and the negative tuples the same way as the sliding window counterpart. However, for the DICO case , union operator will convert its delta input to complete output. For the CIDO case, the union operator will convert its complete input to delta output.

### 3.4.2 Window Difference

Window difference operator is a unique operator, as it is asymmetric and expensive. By the definition of the difference operator, only tuples in S but not in R will be outputted. It is important since it is the only operator to compare the difference between two sets. It is asymmetric which means that processing an input tuple depends on whether the tuples is from input stream S or R. Relation of this is described in the following table:

| may output | input for S | input for R |
|---|---|---|
| + | + | - |
| - | - | + |

This means the tuple arrives in S may cause a tuple with the same sign to be outputted and a tuple arrives in R may cause a tuple with the opposite sign to be outputted. Thus, the operator will produce a premature expiration as a positive tuple $t_1$ has been outputted for the previous window and its timestamp is still valid for the current window. Then a positive tuple $t_2$ with the same attribute as $t_1$ arrives in the R in the current window. This will cause the previously outputted tuple $t_1$ to expire due to operator semantics even it is still in the window. Thus, the delta output of the difference operator will have to represent this kind of change using negative tuples because the next operator cannot determine such premature expiration by scanning its own states. From the table, we can also see that difference operator may flip during a hop if the execution strategy is TAAT. For example, this could happen if a tuple $t$ arrives in S and gets outputted. Later in the same window, a tuple with the same value come in R. This makes the previously outputted $t$ invalid in the for current window. In the next section, we will discuss execution strategies in detail.

In a DIDO case, the difference operator follows the above relation, and processes its input like sliding window difference operator. In a DICO case, although the difference operator will produce premature expiration, it will not produce any output until it can merge all the changes into a complete result. Such complete result will not contain negative tuples. The complete result implicitly suggested that tuples which are not in the current result have expired. In a CICO case, the difference operator will output the current valid tuples as complete result. No incremental

maintenance is needed for the next operator. In a CIDO case, the difference operator will convert its input to delta to explicitly indicate changes. It will compare its current result cache to its previous result cache to determine what tuples to be outputted. Due to its definition, it is likely that the output will have negative tuples.

## 3.5 Window Aggregates and Group-by

The group-by operator maps each input stream tuple to a group and produces one output tuple for each nonempty group G. The output tuples have the form <G, Val>, where G is the group identifier and Val is the group's aggregate value. The aggregate operator can be seen as a Group-by operator with single group. The output of the group-by operator is different from all other operators as the number of tuples is fixed for most Group-by operator, for example AVG, MAX, COUNT. Group-by operator can generate a positive tuple per group per each hop.

# Chapter 4

# Execution Strategy

In this chapter, we will discuss the execution strategy about weak non-monotonic operators and strict non-monotonic operators. Weak non-monotonic operators are operators that will not need to send out negative tuples to expire its previous result. We will use join operator as our example. Strict non-monotonic operators are operators that will always need to send out negative tuples to expire its previous result. We will use difference as out example.

## 4.1 Execution Strategy for Join Operator in Hopping Semantics

A join operator working under TAAT strategy will work on a per tuple base. That is after probing the right state, every input tuple from the left stream is inserted into the left state, and vice versa. This procedure produces join results incrementally. A tuple usually produces partial result in this manner because it can be probed and joined if there are tuples with the same value arrive at the opposite side latter in the same window. The result set for the whole window may contains more tuples

related to this tuples, but we can not determine by now. Thus, the tuple is inserted into the state and waiting to be probed by later input tuples from the opposite side. Since tuples from both sides will be probed by the opposite side, we will have to build hash index on both sides of the states. On the other hand, if a join operator uses WAAT, it will not perform any probing during dequeue. Every tuple is directly inserted into the states. The operator will only use its one side of the state tuples to probe the other side when it has a full window. Thus, unlike the TAAT, when probing, for each tuple, the operator can determine how many tuples are needed to be outputted. So we can build a hash index on one side instead of both sides. By using the WAAT strategy, the join operator saves on building less hash indexes, while paying the price of delayed and fluctuating output.

## 4.2 Execution Strategy for Difference Operator

A difference operator working under TAAT strategy will work on a per tuple base also. Every time the difference operator reads a input tuple with value $v$, it tries to determine whether this tuple should be output or not. However, without seeing all the tuples in the current window, such early decision is likely to be wrong and may have to be corrected latter. On the other hand, if WAAT strategy is adopted by the difference operator, redundant work caused by early decision can be avoided. As one can imagine, when the window is full, the number of tuples having a particular value $v$ is fixed. So the number of tuples to be output is also fixed. There is no going back and forth about whether to output a tuple. The potential savings by adopting the WAAT strategy are much greater compare dto the join operator adopting the same strategy. We will analyze the worst case scenario of TAAT strategy in the following section, and show the experimental results in Chapter 7 .

## 4.3 Cost Analysis of TAAT Strategy

**Definition 2.** *A flip corresponding to the operator changing its decision about outputting a tuple or not outputting it.*

As discussed above, the TAAT strategy forces the operator to make an early decision about whether to output a tuple without having all the information it needs. Similar situation can happen in the sliding window query scenario. However, as the semantics of the sliding window query require updates on the result for any newly arriving tuple. Such flip caused by a single tuple is a valid output in the sliding window query scenario. However, in the hopping window query semantics, flips in a hop are useless and wasted CPU resources. Thus we study the behavior of flips and aim to reduce the number of flips in each hop. As we discussed in Chapter 3, a strict non-monotonic operator is the only kind of operator that will generate flips. We use difference operator as an example here. We first analyze for the set semantics of the difference operator, then present the more complicated bag semantics. According to the states and output of a difference operator, we built a DFA to model the behavior of a difference operator. In order to simplify the discussion, we assume all input tuples have the same value $v$. In general the total number of flips for a difference operator will be the sum of flips of each distinct value. We use this DFA to prove the maximum number of flips can occur for a difference operator giving its inputs equals to the sum of the number of flips in its left input and right input. We make the additional assumptions on the input of the DFA.

- For the same tuple, a positive tuple will always come before the corresponding negative tuple. Such assumption is based on the fact that if we never send out a valid tuple, we will not send out a negative tuple to invalidate it.

- The tuple for stream S will come first, since we want to determine the maxi-

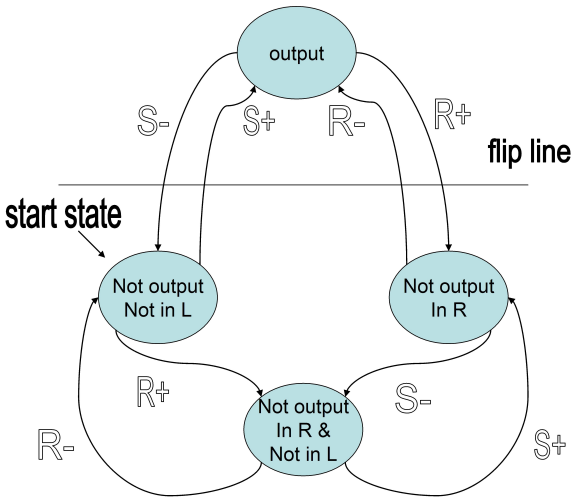


Figure 4.1: flip set semantics

mum possible number of flips. No flips will be triggered if a tuple comes into R first, without canceling any previous output. Such thus case cannot be the worst case.

- Every state is a final state. Because a tuple can be either output or not output.

In Figure 4.1, a flip is a transaction that crosses the flip line. The flip line is the line between the output state and the not output states. Any transaction crossing this line means the operator has changed its mind about whether to output a certain tuple. Let's look at a simple case. All the streams originally do not have any flips. What is the maximum number of flips a difference operator can produce? Let's look at the DFA. We can see 2 flips. In the case that a tuple gets outputted first then a tuple with the same value arrives in the input stream R and canceles the previous outputted tuple. Transaction between "not output" state will not cause flip. Since one input symbol can only cause at most one flip, thus the number of flips will not excess the length of the input. And the input is flips from the S stream and flips from R stream. So for a difference operator in worst case, the number of flips will be equal to the sum of number of flips from input stream S and input stream R. So

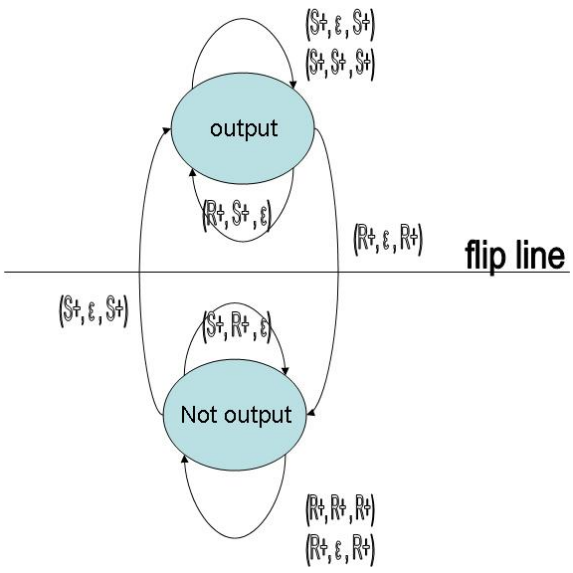


Figure 4.2: flip bag semantics

we can see that, flips are linear additive.

In Figure 4.2, we use a pushdown automaton to capture the bag semantics of the difference operator. Because in bag semantics, we have to keep track of the number of tuples with value $v$ in both sides additional storage space is needed. So PDA is needed. In the PDA, both the output and not output state is the final state. The transactions have the following schema: (input symbol, pop from stack symbol, push into stack symbol). Bag semantics also allow duplicates, multiple tuples with value $v$ may be outputted. So we model the flip a little differently from its set semantics counterpart. We also assume that there are no negative tuples in the input stream. In Figure 4.2, the maximum number of flips equals to the length of the longest sequence of transactions that originated and ended at the not output state. We can see that, flips are linear additive for both bag semantics and set semantics.

# Chapter 5

# Query Optimization

## 5.1 Physical Implementation

In this section, we describe the implementation of each operator.

### 5.1.1 Window Select and Window Project

Under DIDO and CICO cases, both the Select and Project operator are stateless. They process tuples on the fly as they arrive just like their sliding window counterpart. However, when conversion is needed, the DICO case and CIDO case, Select and Project operator become stateful. To convert from Delta to Complete output, depending on the amount of negative tuples, either (First In First Out) FIFO list or hashmap can be use. When there are few negative tuples in the delta Input, FIFO list is better than hashmap. Because after checking the selection condition, new positive tuples will be appended to one end of the list. Old tuples will be expired on the other end of the list. For the occasional negative tuples, if any, a full scan is needed. If the number of negative tuples are large, a hashmap could be used to quickly find the corresponding positive tuples. The select operator will output its

full state as the complete result when the current window is full.

To convert from complete to delta, the operator needs to keep the previous result and the current result in separate hashmaps. For the tuples in the previous result but not in the current result, if the timestamp indicates it is still in the window, a negative tuple must be generated because this is a premature expiration. If the timestamp indicates it is not in the window, a negative tuple will be generated to indicate the premature expiration.

### 5.1.2 Window Join

Nested loop or hash join can be used as join method for the natural join operator. However, nested loop is only suitable for very small window sizes. Thus, we will not discuss the implementation of nested loop here. We used two hashmaps as out operator states. Tuples from stream S and stream R are put into the hashmaps using the join attribute as key to allow quick look up for matched tuples.

In DIDO case, processing a tuple is done in the same way for both input side. Insert, probe and purge have to be done on both sides. Positive and negative tuples are treated the same with the difference in the output sign. Join needs to access previous input tuples which are still fall into the current window while processing the newly incoming tuples. Every tuple is processed in the same way as the sliding window counterpart.

In DICO case, we have to be aware that instead of delta output, complete output is needed. To achieve this semantics, as tuples arrive, Join operator keeps and maintains its previous output by purging out expired and invalid tuples, if any. All the new tuples are processed in the same way as in DIDO. Such maintenance is the

only extra work compare to DIDO.

In CICO case, we know the current output contains everything we need to know for the current window. Operator can simply insert these tuples into the state and perform a join in either TAAT or WAAT fashion. After that, the operator don't have to keep input tuples anymore. Early purge is possible, this is a potential memory saving. Output produced by above description is called the complete output.

In CIDO case, the operator also need extra work to convert complete input into delta output. To do that, previous outputted tuple needed to be stored inside the operator. Current output will have to be compared with the previous output. Whatever is in the current output, but not in the previous output corresponding to the positive tuples for the next operator. Whatever is in the previous output but not in the current output corresponding to the negative tuple for the next operator. Even complete result does not contain negative tuples, the delta output may contain negative tuples as the complete result may from a operator that will cause premature expiration.

### 5.1.3 Window Difference

The difference operator will adopt a WAAT strategy to reduce flips in a hop when complete input is given. We use hashmap as the state of the difference operator. A Key of the hashmap is the concatenation of all the attributes in the input tuples. In the DIDO case as new tuples arrive, the difference operator has to maintain its partial current output up to date by inserting positive tuples into the output and purging out expired and invalid tuples. If the input contains negative tuples the

difference operator will purge out the corresponding positive tuples. In the CICO case difference operator does not keep its previous input, as the input is complete. Input tuples will all be inserted into the states of the operator and perform the difference at once. In DICO and CIDO cases, besides the states, the previous output is also needed for the conversion from complete to delta and vice versa.

### 5.1.4 Window union

Implementation of union operator is very similar to select and project operators. Under DIDO and CICO cases, union operator is stateless. Process input tuples on the fly, merging tuples from two input stream into single output stream. It also becomes stateful in DICO and CIDO cases. Data structures are the same as select and project operators.

## 5.2 Cost Model

Before we discuss our cost model, we introduce the symbols for cost model in this thesis.

### 5.2.1 Join

To simplify the discussion, we assume input are in both complete or both delta. In the case of one is delta and the other one is complete, we can convert them into the same format. We focus on CPU cost in this cost model.

The cost for performing a join over Stream A and Stream B is:

CPU Cost for A join B = Insert + Probe+ Concatenation+Purge

DIDO:

Table 5.1: Symbol

| term | meaning |
|---|---|
| $\lambda_S$ | arrival rate for stream S, S=A,B,C... |
| $\lambda_{dpI}$ | positive delta tuple arrival rate for stream I, I=A,B,C... |
| $\lambda_{dnI}$ | negative delta tuple arrival rate for stream I, I=A,B,C... |
| $\lambda_c$ | complete result tuple arrival rate for stream I, I=A,B,C... |
| $\sigma_{IJ}$ | selectivity between stream I,J, I $\neq$ J, I=A,B,C... J=A,B,C... |
| $C_{lookup}$ | constant for hash lookup |
| $C_{insert}$ | constant for insert a tuple into hashmap |
| $C_{concate}$ | constant for concatenating two tuples |
| $C_{select}$ | constant cost for selecting one tuple. |
| $C_{project}$ | constant cost for projecting one tuple. |
| $C_{union}$ | constant cost for union operator to process one tuple. |
| $C_{Group-by}$ | constant cost for group-by operator to process one tuple. |
| $\|S_I\|$ | state size for stream I, if the operator is stateful. |
| $\|W_I\|$ | window size for stream I |
| $\|H\|$ | hop size |

Insert=$C_{insert} * |H_A| * \lambda_{dpA} + C_{insert} * |H_B| * \lambda_{dpB}$

Probe= $C_{lookup} * |H_A| * \lambda_{dpA} + |C_{lookup} * H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} + C_{lookup} * |H_B| * \lambda_{dnB}$

Concatenation= $C_{concate} * |H_A| * \lambda_{dpA} * \sigma_{AB} * |S_B| + C_{concate} * |H_A| * \lambda_{dnA} * \sigma_{AB} * |S_B| + C_{concate} * |H_B| * \lambda_{dpB} * \sigma_{AB} * |S_A| + C_{concate} * |H_B| * \lambda_{dnB} * \sigma_{AB} * |S_A|$

Purge= $C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dnB} + (\lambda_{dpA} - \lambda_{dnA}) * |H_A| * C_{purge} + (\lambda_{dpB} - \lambda_{dnB}) * |H_B| * C_{purge}$

The insertion cost includes putting the tuples into the hashmap. The amount of tuples needed to be put in to the hashmap is the tuples arrive for the hop. Only positive tuples needed to be inserted into hashmap because after a negative tuple cancel the corresponding positive tuple, itself also reach the end of the journey.
The probing cost is the cost of hash look up for every input tuple. Both positive and negative needed to probe the other side of the input to determine whether

matched tuples can be found. The concatenation cost is the cost to concatenate the probing tuple and those matched tuples. On average, The number is equal to the selectivity multiply by the State size. The last one is purging cost. It includes purge by negative tuples, and state scan purge(by timestamp). On average, this is equal to the the number of input tuples minus the number of negative tuples.

DICO:

Insert=$C_{insert} * |H_A| * \lambda_{dpA} + C_{insert} * |H_B| * \lambda_{dpB}$

Probe= $C_{lookup} * |H_A| * \lambda_{dpA} + |C_{lookup} * H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} + C_{lookup} * |H_B| * \lambda_{dnB}$

Concatenation= $C_{concate} * |H_A| * \lambda_{dpA} * \sigma_{AB} * |S_B| + C_{concate} * |H_A| * \lambda_{dnA} * \sigma_{AB} * |S_B| + C_{concate} * |H_B| * \lambda_{dpB} * \sigma_{AB} * |S_A| + C_{concate} * |H_B| * \lambda_{dnB} * \sigma_{AB} * |S_A|$

Purge= $C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dnB} + (\lambda_{dpA} - \lambda_{dnA}) * |H_A| * C_{purge} + (\lambda_{dpB} - \lambda_{dnB}) * |H_B| * C_{purge}$

Conversion=$C_{insert} * |H_A| * \lambda_{dpA} * \sigma_{AB} * |S_B| + C_{lookup} * |H_A| * \lambda_{dnA} * \sigma_{AB} * |S_B| + C_{insert} * |H_B| * \lambda_{dpB} * \sigma_{AB} * |S_A| + C_{lookup} * |H_B| * \lambda_{dnB} * \sigma_{AB} * |S_A| + (\lambda_{dpA} - \lambda_{dnA}) * |H_A| * C_{purge} + (\lambda_{dpB} - \lambda_{dnB}) * |H_B| * C_{purge}$

Insert, probe and purge are the same as the DIDO case. However, it has a extra conversion cost. It has to update its result from previous window using the delta result it just generated. Positive tuples will insert into the previous result and negative tuples will remove from the previous result. State scan purge may also needed if mature expiration is not indicated by negative tuples.

CICO:

Insert=$C_{insert} * |W_A| * \lambda_{dpA} + C_{insert} * |W_B| * \lambda_{dpB}$

Probe= $C_{lookup} * |W_A| * \lambda_{dpA}$

Concatenation= $C_{concate} * |W_A| * \lambda_{dpA} * |W_B| * \lambda_{dpB} * \sigma_{AB}$

Purge=0

Insertion cost for CICO equals inserting all the tuples belongs to current window into the state. Probeing cost is equal to use every tuple in one side of the stream to probe the other side. In our case tuples in Stream A to probe tuples in Stream B. Concatenation cost is the same as DICO case. There is not purging cost because operator have the complete result as input.

CIDO:

Insert, Probe, Concatenation and Purge cost are the same as CICO. However, there is a additional cost for converting the complete result into the delta result.

conversion= $|W_A| * \lambda_{dpA} * |W_B| * \lambda_{dpB} * \sigma_{AB} * 2(C_{lookup})$

That is the operator uses the previous result to probe the current result to determine what tuples should be remove. It also uses the current result to probe the previous to determine what tuples should be added.

### 5.2.2 difference

The cost for performing a Difference over Stream A and Stream B is:

Cost = Insert + CrossProbe+ SelfProe +Purge + Conversion

Purge and conversion are optional sometime.

DIDO:

Insert=$C_{insert} * |H_A| * \lambda_{dpA} + C_{insert} * |H_B| * \lambda_{dpB}$

CrossProbe= $C_{lookup} * |H_A| * \lambda_{dpA} + C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} + C_{lookup} * |H_B| * \lambda_{dnB}$

SelfProbe=$C_{lookup} * |H_A| * \lambda_{dpA} + C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} + C_{lookup} * |H_B| * \lambda_{dnB}$

Purge=$C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dnB} + (\lambda_{dpA} - \lambda_{dnA}) * |H_A| * C_{purge} + (\lambda_{dpB} - \lambda_{dnB}) * |H_B| * C_{purge}$

Conversion= $|H_A| * \lambda_A \sigma_{AB} * (C_{lookup}) + 1/2 * |H_A| * \lambda_A \sigma_{AB} * (C_{insert})$

Only the newly arrive positive tuples will get inserted into the states. Then every intput tuple will probe the other side of the stream and the same side. To determine whether to output itself or not. Purge cost is the same as Join operator under DIDO case. conversion is needed because we need to maintain the partial current result.

DICO:

Insert=$C_{insert} * |H_A| * \lambda_{dpA} + C_{insert} * |H_B| * \lambda_{dpB}$

CrossProbe= $C_{lookup} * |H_A| * \lambda_{dpA} + C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} +$ $C_{lookup} * |H_B| * \lambda_{dnB}$

SelfProbe=$C_{lookup} * |H_A| * \lambda_{dpA} + C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dpB} + C_{lookup} *$ $|H_B| * \lambda_{dnB}$

Purge=$C_{lookup} * |H_A| * \lambda_{dnA} + C_{lookup} * |H_B| * \lambda_{dnB} + (\lambda_{dpA} - \lambda_{dnA}) * |H_A| * C_{purge} +$ $(\lambda_{dpB} - \lambda_{dnB}) * |H_B| * C_{purge}$

Conversion= $|H_A| * \lambda_A \sigma_{AB} * (C_{lookup}) + 1/2 * |H_A| * \lambda_A \sigma_{AB} * (C_{insert})$

CICO:

Insert=$C_{insert} * |W_A| * \lambda_{dpA} + C_{insert} * |W_B| * \lambda_{dpB}$

CrossProbe= $C_{lookup} * |W_A| * \lambda_{dpA}$

SelfProbe=$C_{lookup} * |W_A| * \lambda_{dpA}$

There is not purging cost as the the next complete input will provide enough information do computation. CIDO:

Insert=$C_{insert} * |W_A| * \lambda_{dpA} + C_{insert} * |W_B| * \lambda_{dpB}$

CrossProbe= $C_{lookup} * |W_A| * \lambda_{dpA}$

SelfProbe=$C_{lookup} * |W_A| * \lambda_{dpA}$

Conversion= $|W_A| * \lambda_{dpA} \sigma_{AB} * (C_{lookup})$

It just need the extra cost of conversion to output delta result.

### 5.2.3 Select, Project

Since select , project is a stateless operator. We ignore the result conversion in here because there is no need for stateless operator to do such conversion. Operator below select and project can produce either the delta result and complete result for the operator above select or project.

**Cost for select over Stream A:**

Cost $\sigma_A$=$\lambda_A * C_{select}$

**Cost for project over Stream A:**

Cost $\pi_A$=$\lambda_A * C_{project}$

### 5.2.4 Group-by

**Cost for Group-by over Stream A:**

As stated in earlier, the way group-by operator processes tuples is different from other operator. The cost dependents on the typed of kind of aggregate used . However, for most group-by operator, operator only has to access one column per tuple to compute the aggregate result, this is the major cost for group-by operator. Base on the above assumption, if the aggregate function is not incremental computable, them we recompute it every time the window moves. Thus we will always prefer complete result input. If the input is not complete result, the operator can convert it into a complete result. Thus there is not state maintenance cost. Even the function is incremental computable, the group-by operator has to process every tuple so cost of group-by operator is Cost(OPgroupby)= $\lambda_A * |W| * C_{group-by}$

# Chapter 6

# Mode Assignment and Optimal Plan Generation

Giving the above cost model, we can use dynamical programming to generate all possible plans and plug in the cost model to determine which plan is the optimal plan. As we mentioned earlier, the representation of intermediate results can greatly affect the cost of the plan. Thus, we now propose to incorporate mode assignment into the plan enumeration process as an important step to get a truly optimal plan. As we stated before, each operator has four combinations of input and output modes. A mode assignment for a operator is to choose a suitable input and output mode out of those four combinations. A mode assignment for a plan is that the choice of a mode for each operator in the query plan. The input mode for the leaf operator in a query plan is the input mode for the plan, typically, they tend to be delta input. The output mode for the root operator is the output mode of the query plan. Inside a query plan, there are many different possible mode assignments. The mode assignment with minimum cost that conforms with the input mode and output mode of the plan as a whole is called optimal mode assignment. The plan is called optimal

assigned plan.

## 6.1 Mode Assignment

**Theorem 1** (Theorem of Optimality)**.** *If a given plan P has a optimal assignment A, then any sub-assignment $A_{sub}$ of sub-plan $P_{sub}$ of P starting from the leaves is also optimal.*

*Theorem of Mode Assignment Optimality.* In the following proof, we assume root node has depth 1. For the sake of contradiction, suppose plan $P$ with k depth consists operators $O_1....O_n$ is optimal. But for a certain sub-plan, plan $P_{sub}$ up to depth j is not optimal, chose another mode assignment for the operators in $P_{sub}$ can achieve a plan with low cost. Thus cost for $P_{sub}$ is larger for $P_{subReAssign}$. If this is the case, we can substitute the plan $P_{sub}$ with the new assigned $P_{subReAssign}$ into plan $P$ to attain a better plan. But this contradicts the plan $P$ being optimal. Thus, plan $P_{sub}$ has to also optimal. □

Base on the above theorem, we can conclude that by choosing optimal assignment at each depth of the plan, our plan assignment algorithm will always produce optimal assignment. In the algorithm, *assignMode()* is a helper functions that calculate the cost of each input and output combination then pick the cheapest one amount those output complete result and delta result. *getMode()* is the recursive function that traverse the plan.

**Algorithm 1** getMode (queryplan represented by root node)

```
if S is a leaf node then
  assignMode(S)
else
  for all S_i ⊂ S.children do
    getMode(S_i)
  end for
  assignMode(S)
end if
```

## 6.2 Optimal Plan Generation By Dynamic Programming

From the above theorem, we know that, in order to find the optimal mode assignment, one does not have to consider all $4^n$ possible combinations, as many of them will not contribute to the optimal plan. A greedy approach with time complexity $O(2n)$ is enough to determine the optimal mode assignment in a giving plan. $n$ is the number of operators in the giving plan. Giving this, we can now incorporate the mode assignment into dynamic programming. [23] first used dynamic programming in query optimization, then [15] extended the idea and used iterative dynamic programming to consider bushy plan. We will give a skeleton description of our algorithm; reader interested in further study dynamic programming can look at [23] [15].

Dynamic programming considers all possible ways to join the streams. First, it considers all two-way join plans by using the stream as building blocks and calling the *joinPlans()* function to build a join plan from these building blocks. From the two-way join plans and the streams, dynamic programming then produces three-way join plans. After that, it generates four-way join plans by considering all combinations of two two-way join plans and all combinations of a three-way join plan with an access plan. In the same way, dynamic programming continues to produce five-way,

**Algorithm 2** DiffDP(extended Join Graph)

```
Plans= ⊘
if RootGraph has difference source then
  for i = 0 to n − 1 do
    if S_i is a difference source then
      for i = 0 to n − 2 do
        for all S ⊂ S_0...S_{i−1}, S_{i+1}...S_{n−1} such that |S| = i do
          Remove S from RootGraph, ADD S to LeftChild and RightChild of
          S_i
          DiffDP(S_i .LeftChild)
          DiffDP(S_i .RightChild)
          Plans.AddPlan (DP(RootGraph))
          Restore S
        end for
      end for
    end if
  end for
else
  Plans.AddPlan(DP(RootGraph))
end if
PickOptimalPlan(Plans)
```

**Algorithm 3** DP(extended Join Graph)

```
for i = 2 to n do
  for all S ⊂ R_1, ..., R_n such that |S| = i do
    optPlan(S)= ⊘
    for all O ⊂ S do
      optPlan(S)=getMode( optPlan(S) )∪ getMode(joinPlans (optPlan(O),
      optPlan(S\O)))
    end for
    prunePlans(optPlan(s))
  end for
end for
prunePlans(R_1, ..., R_n)
```

six-way join plans and so on up to n-way join plans. The *prunePlans()* function discards unneeded plans. Pruning is possible because the A $\bowtie$ B plan and the B $\bowtie$ A plan do the same work, only the cheaper plan of these two is needed if we consider asymmetric data structure. Optimal plans would be retained in *optPlan* (A, B).

The idea to incorporate mode assignment into Dynamic programming is that instead of generating all possible plans of size one, than use those to generate plans of size two, etc..., we will first generate all possible plans of size one with mode assignment. This will cost the algorithm to generate twice the plan at each level (size one, size two ...) Just total time complexity is $O(3^{2n}) = O((9^n)$ compare to traditional DP which is $O(3^n)$ [24][21].

Our algorithm 2 considers all possible position in a query plan for difference operators. The input of the algorithm is a plan with difference operators all the way pushed to the bottom. As the algorithm traverse the search space, it removes join operators from the root graph and put them into the children of difference operator. After each removal, it has to restore those removed operators so that they can be removed with another set of operators in a later stage and put into the children of difference operator again.

## 6.3 Heuristics Rule

We can see that to relay on pure dynamic programming is not practical for real application. We will have to consider heuristic to improve the performance of query optimization.

We employ the following heuristic as difference pushdown rule. The rule is that always push difference operator to the depthest level in a given query plan. Over

our DiffDP experiment, over 98 % of the time, the plan with difference operator pushed to the lowest possible position in the plan is the plan with the lowest cost.

# Chapter 7

# Experimental Evaluation

## 7.1 Overview

In this chapter, we demonstrate our techniques in a real stream data processing engine. We have implement the multi-mode-aware hopping query process method using Sun Microsystem JDK 1.5.0. Testing was performed on a Linux cluster. Each node has 4 CPU with 1000Mhz each and a total of 4 GB of memory. We use one of the nodes as stream generator, an other nodes as the processing engine. There are following objectives of our experiments:

- Validate the cost models by comparing the execution times of various hopping window queries between cost models and the prototype program.
- Examine the performance trends of the four different input and output combinations for each individual operator.
- Examine the performance trends and identify the best mode assignment in relation to the parameter values.

- Demonstrate the effectiveness of the difference push down heuristic compare to dynamic programming .
- Demonstrate the performance gain by combining our heuristic and mode assignment.

Inputs to the processing engine are data streams generated using a data generator. The data generator generates stream data sets as a sequence of tuples. Inputs to the data generator are the number of tuples in the data set, the number of attributes in the stream schema, the number of distinct values of each attribute in the schema, the stream rate (number of tuples per second) and its distribution . Each tuple has a timestamp attribute, whose value is determined based on the stream rate and its distribution. Mean interval between two tuples is 1 millisecond. Values of each attributes are assigned randomly with the uniform distribution.

## 7.2 Query 1: Single Join Operator Plan

We will test 10 variations of Query 1, with hop/window ratio from 10% to 100% for all 4 combinations of DIDO, DICO, CICO, and CIDO. The Figure 7.1 clearly shows that when ratio is small, delta input is preferred by Join operator. DIDO is the least expensive mode for this case because it does not do any redundancy computation. DICO trailing a little behind because it has overhead to convert incremental output to complete output. CICO is the third because it has to compute lots of redundant tuples. CIDO is the most expensive one because not only it has to compute redundant tuples, but also convert them into delta result. We can also see that when the ratio become larger and larger, the complete result input has more and more advantages over the delta input. Both the experimental run and the cost model show these trends. We can conclude that when window is not overlapping at

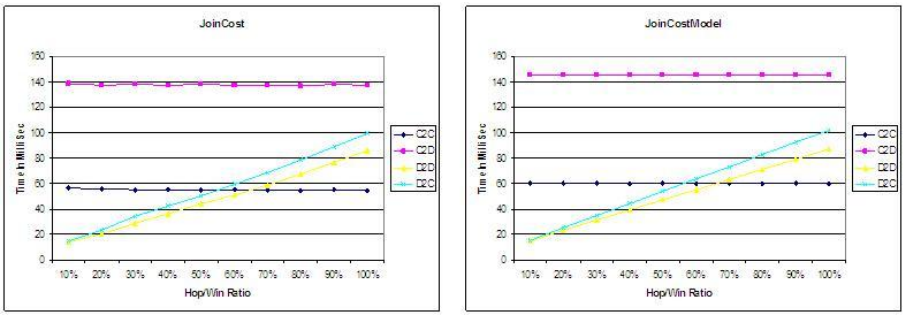


Figure 7.1: Query 1: Single Join Operator Plan

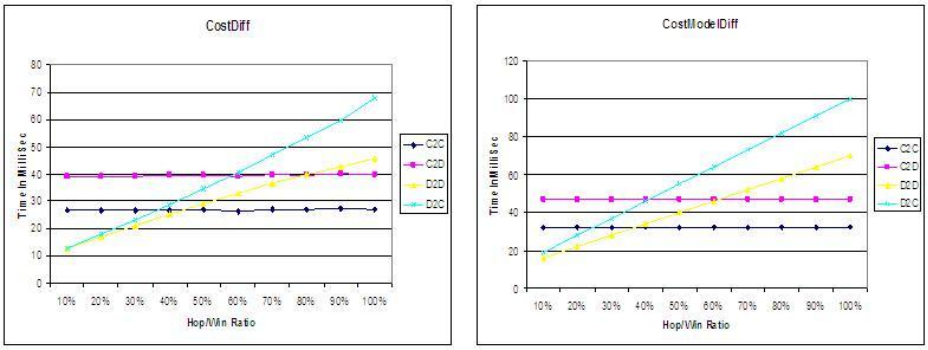


Figure 7.2: Query 2: Single Difference Operator Plan

all, using CICO can save up to 34% compared to DIDO in our setup.

## 7.3 Query 2: Single Difference Operator Plan

We will again test 10 variation of Query 2, with hop/window ratio from 10% to 100% for all 4 combinations of DIDO, DICO, CICO, and CIDO. The Figure 7.2 shows that although the delta input is still preferred by Difference operator when ratio is small, the gap between processing delta input and complete input is smaller than the join counterpart. The reason for that is flipping inside the difference operator. The trends are similar to the join operator, both the experimental run and the cost model show that. When window is not overlapping at all, using CICO can save up to 40% compared to the DIDO in our setup.

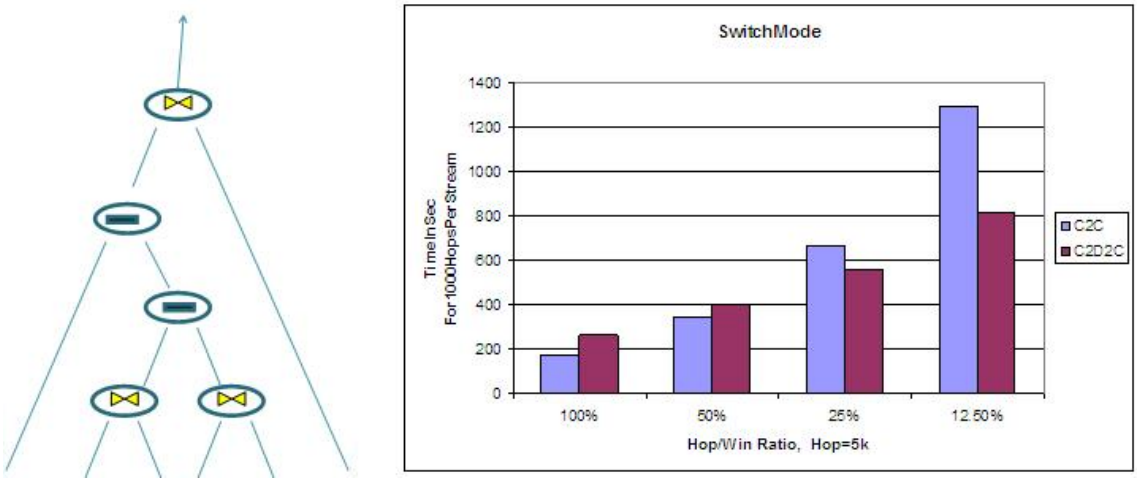


Figure 7.3: Query 3: Ratio Triggered Mode Switch

## 7.4 Query 3: Ratio Triggered Mode Switch

In this experiment, we will vary the window size and fixed the hop size in this experiment. Hop size is fixed at 5000 ms. We compare the actual run time of generic complete input complete output plan and the assigned plan. In the assigned plan, we assigned the last difference operator with CIDO mode and the last join operator with DICO mode. In Figure 7.3, we can see that as we increase the window size from five thousand millisecond to forty thousand millisecond, switching to delta internally is a good choice when hop/window ration is small.

## 7.5 Heuristic Rule vs Exhaustive Search

In this experiment, we randomly generate plans with less than 8 operators. Then we randomly assign different selectivity for each operator. We use this as our input to the exhaustive search optimizer. Figure 7.4 shows that over 98% of the time, the plan returned by our heuristic was as good as the plan return by the exhaustive search optimizer. In Figure 7.5 shows a actual runs of the difference operator push

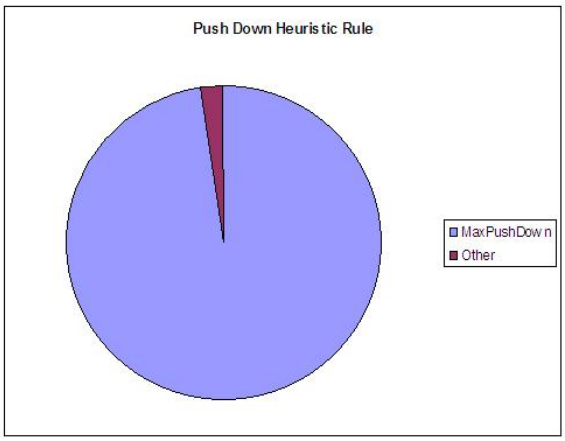


Figure 7.4: Pushdown Heuristic

down plan compared to difference pull up plan.

## 7.6 Putting It All Together

In this experiment, we compare the performance of three different query plans.

- The user default plan (D2DPullUp)
- The plan with heuristic(D2DPushDown)
- The mode assignment plan with heuristic (D2C2DPushDown)

The user default plan is a plan with incremental input and incremental output. The plan with heuristic is a plan attained by applying our heuristic. The mode assignment plan with heuristic is a plan attained by running the mode assigner on the plan generated by heuristic. In Figure 7.6, we can see that, D2DPullUp has the worst performance, It has not taking advantage of that the difference operator push down can reduce the amount of work the up most join operator has to perform. D2DPushDown has been taking the advantage of pushing difference, however, it is not as efficient as D2C2DPushDown because D2C2D using complete result between

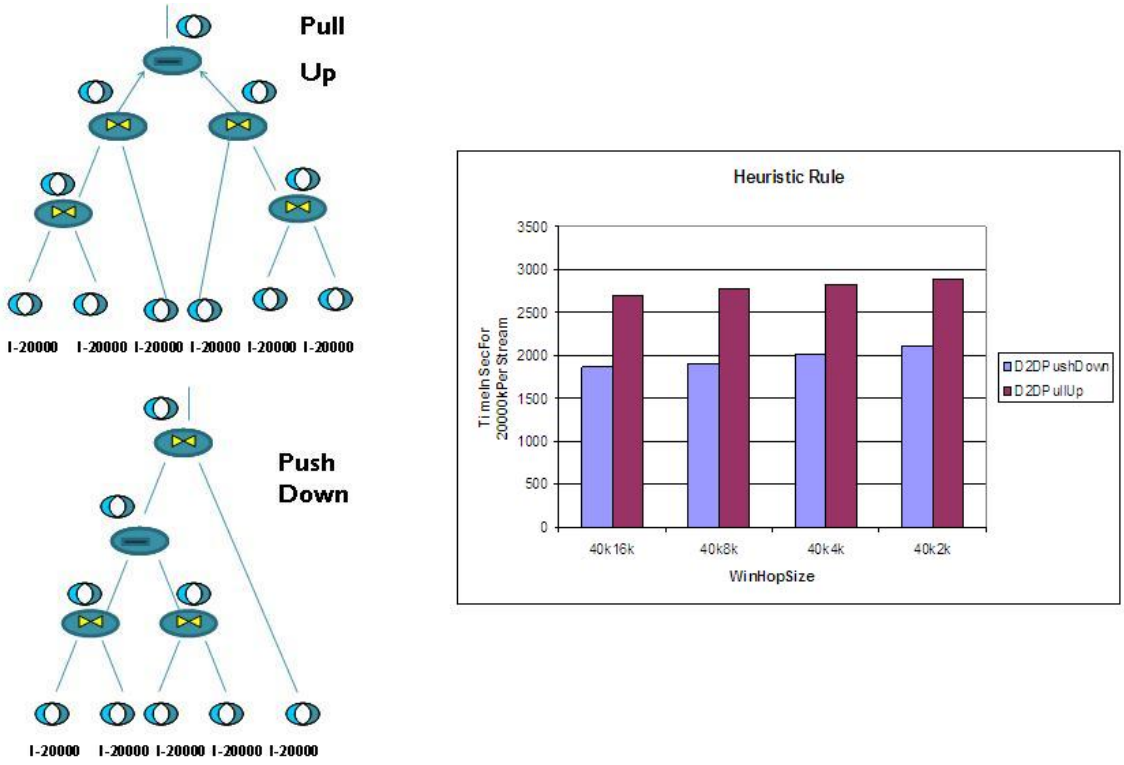


Figure 7.5: Pushdown vs Pullup

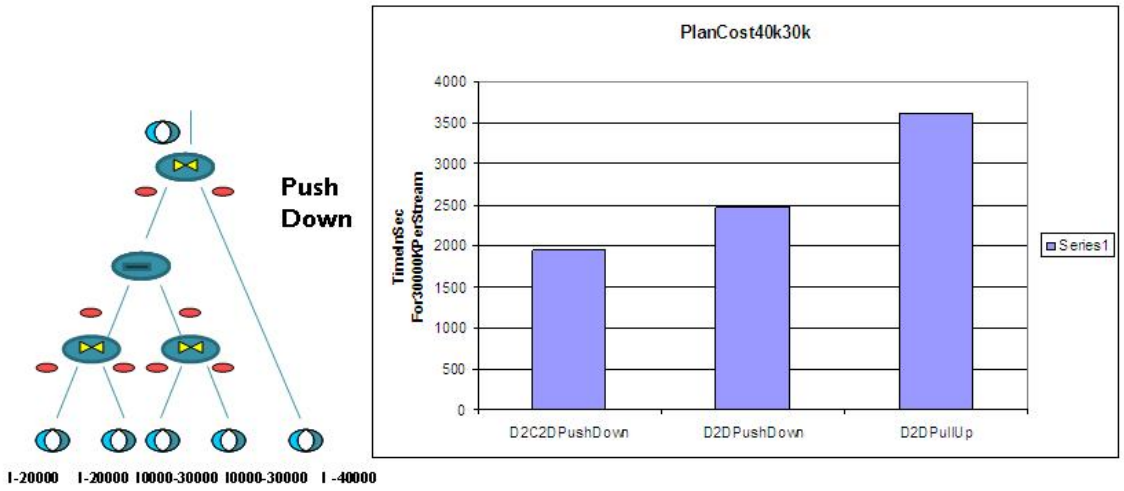


Figure 7.6: All Together

operators internally, so that the operators do not have to maintain its states. By using complete result, the stream engine can also avoid the overhead of processing negative tuples which generated by difference operator.

# Chapter 8

# Related work

In this section we discuss the related work in stream processing area.

Most stream query processing research over the past few years has focused on sliding window queries. Cost models and optimization techniques have been developed over the last few years [6][20]. However, they may not work well for hopping window queries.

Continuous query evaluation is a hot area of research in the database community. There has been some initial work of investigating how to process data efficiently while handling negative tuples. [13] presented several execution strategies for sliding window implementation with negative tuple support. Their work on negative tuples mainly focuses on tuples that expire maturely (fall out of window) and can not work well when larger amount of tuples expire prematurely (false to satisfy operator semantics). [10]further investigates incremental evaluation technique over sliding windows and minimizes overhead for having negative tuples in the system. However, their negative tuples again are employed to capture the case when tuples expire maturely. Besides that, in the hopping window query scenario, using reevaluation

may be a better choice. Our work considers both the incremental and reevaluation technique. We present a hybrid approach that combines these two techniques within one integrated query plan to attain a steady output rate and minimize system overhead. Unlike previous work, we considering negative tuples approach for both mature and premature expiration.

[22] presents join reordering technique that works for not only natural join but also antijoin. Their work is different from ours as their work is base on traditional relational databases and did not consider the streaming environment and the effect of negative tuples.

[16] study the hopping window semantic and memory sharing of aggregate queries. Their technique focus only on aggregate queries and can not easily be transfered to other kind of queries. Lastly, our work is related to [12] in the sense that we both try to look for a patten to efficiently update the query results. [12] has presented a classification of update patterns of continuous queries and applied it to solve two problems: 1) defining precise semantics of continuous queries with a clearly defined role of relations and their update patterns, and 2) efficient query execution over sliding windows. We investigated their classification. Instead of only using negative tuples, we will use both the complete result for each window and negative tuples methods to present the premature expiration and change the execution strategy accordingly.

# Chapter 9

# Conclusion and Future Work

## 9.1 Conclusion

In this thesis, we address the problem of processing continuously hopping window queries. We propos to use both delta and complete results as a intermediate representation between operators to reduce processing cost. Given a query plan, by assigning each operator a suitable processing mode, we can find an optimal mode assignment for the plan. This can be done by running our mode assigner. We develop analytical cost models and validate them through experiments. We also empirically studied the efficiencies of our heuristic and show the case of different assignment and plan shapes with respect to stream statistics. Complete result and incremental result has been studied extensively in different areas of computer science but not in data stream processing with continuously hopping window queries. To our best knowledge, this is the first work to utilize both complete result and delta result in an integrated plan. The results of our experiments indicate that by carefully choosing modes for each operator, we can gain a lot in query performance. In our work, we considered query rewrite and cost models as two essential steps in

query optimization. Applying mode assignment and query rewriting heuristics will have a better performance than applying either of them alone.

## 9.2 Future Work

We intend to pursue the following directions in our future work.

- Tackle other join semantics. Currently, we studied the performance of natural join, which is the most important and common operator. However, there are other semantics of join operator such as outer join for example. The join condition can also vary, one can join tuples from two streams using larger or equal to condition instead of only using equal.

- Consider other join implementation. For example, nested-loop or merge-sort implementation are other possible join implementation. They has their own advantages too. nested-loop is is best for handle unequal condition. Output of merge-sort are sorted in the order of join value.

- Using other query optimization techniques. Currently, we have considered dynamic programming and heuristic rules. However, there are many algorithms that run in $O(nlog_n)$ time and produce decent execution plans.

- Introduce adaptive hopping. Currently, we fixed out hop size for the whole current plan during execution. We plan on removing this constraint. So the system can update the user more frequently if it has extra processing power and update the user less frequently if it is currently busy.

- Conduct an in depth study for each operator. For example, we can study what is the best implementation of each aggregate functionality.

# Bibliography


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

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal The International Journal on Very Large Data Bases*, pages 121–142, 2006.

[5] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.

[6] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.

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

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.

[9] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD Conference*, pages 13–24, 2001.

[10] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. Knowl. Data Eng.*, 19(1):57–72, 2007.

[11] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[12] L. Golab and M. T. Ozsu. Update-pattern-aware modeling and processing of continuous queries. In *ACM SIGMOD Conference*, pages 658 – 669, 2005.

[13] M. A. Hammad, W. G. Aref, M. J. Franklin, and et al. Efficient execution of sliding-window queries over data streams. Technical Report 03-035, Purdue University, 2003.

[14] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[15] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[16] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[17] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322, 2005.

[18] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *ICDCS Workshops*, page 67, 2007.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[20] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. *GeoInformatica*, 9(4):343–365, 2005.

[21] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[22] J. Rao, B. G. Lindsay, G. M. Lohman, H. Pirahesh, and D. E. Simmen. Using eels, a practical approach to outerjoin and antijoin reordering. In *ICDE*, pages 585–594, 2001.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[24] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD Conference*, pages 35–46, 1996.

[25] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, pages 9–18, 2002.