Operator Algorithm Design For Speculative Stream Processing

Jonathan Goldstein
Microsoft Research
Redmond, Washington, USA
jongold@microsoft.com

Mingsheng Hong
Cornell University
Ithaca, NY 14853, USA
mshong@cs.cornell.edu

ABSTRACT

Current stream engine semantics and algorithms assume that input events arrive in timestamp order. This results in well defined operator behavior and efficient processing algorithms. In the case where events may arrive out of order, current systems assume buffering combined with punctuations to disambiguate the arrival order as a preprocessing step.

In previous work: ("Consistent Streaming Through Time: A Vision For Event Stream Processing", CIDR 2007), we introduced a new approach for processing out of order events. More specifically, we introduced a model for event processing which allows speculative output as a means of unblocking operators. As processing continues, incorrect speculative output is retracted and replaced with correct output. This yields a spectrum of query behavior where two processing attributes may be independently varied. The first is the maximum blocking time before the system is forced to speculate. The second is the maximum time for which events are "remembered" for the purpose of correcting previous results.

In this work, we realize our previous vision by formally identifying both necessary properties and conflicting desirable properties that speculative stream processing algorithms may have. Note that since all algorithms which converge to the correct answer over time are considered correct, some of these tradeoffs involve varying the actual output. This leads to an interesting design spectrum for such algorithms.

In addition, we present algorithms which represent interesting points on the design spectrum, and relate them back to these tradeoffs and properties. Finally, we show through a complexity study that the algorithms are practical in that they have near linear-time complexity in the size of input and output streams.

1. INTRODUCTION

In conventional database systems, queries are issued against static snapshots of data, where the coordination of queries and changes to these snapshots is done by the database system. This coordination ensures atomicity and serialization, such that all transactions, which are internally serial in nature, are strictly ordered and the resulting behavior honors that order.

This is in sharp contrast to streaming systems, where queries are issued over changes to data, and are therefore non-terminating. Furthermore, The actual changes to the data are committed prior to arrival at the stream processing system. The streaming system is simply notified of the change along with a commit timestamp.

Queries over streams are, therefore, inherently temporal in nature, and often include windowing constructs, such as "Give me the moving average of process CPU utilization using context switch events on a desktop computer." Various algebras including such termporal constructs have been devised for formulating such queries [1, 6, 4], and the community is converging on approaches based on view update semantics, where output streams are views over input streams, and these views are incrementally maintained [5, 1, 4].

While there is a convergence on query semantics, little attention has been paid to the effect of out of order delivery [10]. More specifically, the only complete approaches to date involve buffering all input events to the stream engine, until a guarantee can be made that all input up to a certain time is received [11, 2]. As such, all stream operator algorithms can assume that all data arrives in order, which greatly simplifies the design of operator algorithms.

In [4], we proposed another alternative: Rather than block all stream processing activity until a guarantee may be established on input completeness, output a speculative answer, with the understanding that it may be revised over time. When a guarantee may be established on input completeness, use this guarantee to establish permanence of the output up to a point in time. [4] further suggests that each query plan allow the specification of 2 parameters: the amount of time a query blocks before forcing speculative output, and the amount of time a query remembers past input for the purpose of revising output. This leads to the consistency spectrum described in Figure 1.

There are some interesting things to note about this Figure. First, the spectrum is a triangle since it doesn't make sense to block longer than you remember. Second, the diagonal of the line is the line of no speculation. Also, the rightmost leg of the triangle is the line of eventual correctness. All points along that leg produce correct eventual output. Finally, the bottom leg of the triangle is the line of zero latency.

While [4] introduced the concepts related to speculative stream processing, identified a temporal stream model, query algebra, and formally defined consistency levels, it left to future work the operator algorithms which implement this consistency spectrum. It also left unanswered what interesting other properties such algorithms might have.

The goal of this work is to provide a well-grounded, viable starting point for the design of such algorithms. Specifically, we formally identify other desirable properties for these algorithms. In-

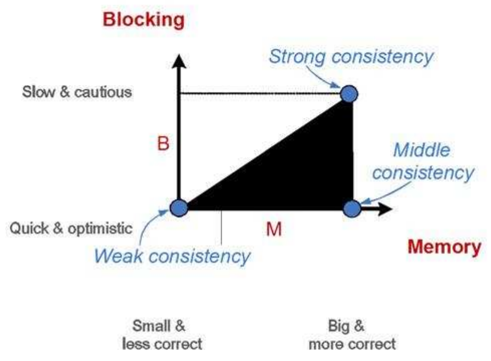

Figure 1: Consistency Spectrum

terestingly, some of these properties are mutually exclusive, and lead to further design choices. We also give precise, provably correct stream processing algorithms for a sophisticated operator (Aggregate) for a particular design point. Furthermore, we sketch how these algorithms could be modified for some alternate design points. Finally, we study the asymptotic efficiency of these algorithms and show that they are efficient even in the worst case.

While the algorithms given in this paper implement solutions very close to the middle level of consistency, including a sketch for the true middle consistency level algorithm, all other levels of consistency may be realized with the additional use of two other operators: Align, which controls blocking time, and Finalize, which controls memory. Align controls blocking time by buffering events in a manner similar to [10] for the purpose of ordering input. Unlike [10], Align may unblock output before a guarantee is available, resulting in potential out of order input and speculation downstream. Finalize controls forced expiration by issuing false guarantees to downstream operators in order to get them to release state prior to a real guarantee. For space constraints, these two operators are described in our technical report [7]. We, therefore, do not need algorithms for other consistency levels to fully realize the proposed consistency spectrum for the operator presented here.

The paper is organized as follows. Section 2 reviews the temporal stream models, and introduces the concepts necessary for later discussions on the operator algorithms. Section 3 presents the operator algorithms, the correctness proofs and the complexity analysis. Section 4 surveys related work. Finally, conclusions and future work are presented in Section 5.

2. PRELIMINARIES

2.1 Temporal Stream Models

In this paper, we assume a discrete time model with a time line composed of chronons [8]. To simplify the presentation, we set the size of a chronon to be 1 time unit. We therefore use natural numbers to represent timestamps.

Unitemporal Stream Model. The stream model adopted by the Stanford STREAM system, as well as many other existing stream systems, assumes, for each event, a timestamp belonging to an application domain. Since the event contains only one notion of time, *application time* [9], we refer to this stream model as *unitemporal*. In addition to the timestamp attribute, each event also contains a fixed set of attribute values, thus conforming to a relational schema. We refer to these attribute values as the *payload* component of the event.

In traditional unitemporal stream models, it is well-understood that each stream is a time-varying relation [1]. Each event is an *insert* into the underlying relation of the stream. A unitemporal stream is append-only with respect to the application time: an event with application time t indicates that it is in the underlying relation starting at time t. We define the *valid interval* $[V_s, V_e)$ of an event to be the application time during which this event remains in the underlying relation. The schema of a unitemporal stream can then be represented by (V_s, V_e; P), where V_s and V_e are respectively referred to as *valid start time* and *valid end time*, and P denotes the relational schema of the payload component. In traditional unitemporal stream models, the attribute V_e is redundant, since each event has a V_e value of ∞. As we will see later, V_e becomes more interesting in the bitemporal stream model.

One major limitation in the unitemporal stream model is that it is not able to model out of order event delivery [10], a phenomenon that often occurs when the stream sources send events to the stream engine via a network that does not guarantee ordering in packet delivery (e.g. if UDP protocol is used). This limitation is addressed by the bitemporal stream model, which we introduce as follows.

Bitemporal Stream Model. The bitemporal stream model supports application time, as well as a second, independent notion of time, local processing server time, called *CEDR time* [4]. A bitemporal stream is append-only with respect to the CEDR time. That is, events are appended to the stream with strictly increasing CEDR time values. For a CEDR bitemporal stream S, *S up to (CEDR time) T* consists of the set of events in S with CEDR time values no more than T. S encodes a relation, called a *virtual relation* R, whose content will be defined based on the events in S (see below). R is time-varying with respect to CEDR time. For each CEDR time T, R_T denotes the content of R associated with stream S up to T. Each R_T itself is time-varying with respect to the application time. That is, each R_T can be viewed as a unitemporal stream.

In unitemporal stream models, "event" and "tuple" are often interchangeable terms. To disambiguate the terminology in the bitemporal stream model, we use "event" to refer to a tuple in a bitemporal stream, and "tuple" to refer to a tuple in its virtual relation.

There are three types of events in the bitemporal stream model. The first type is an *insert*, which inserts a tuple into the virtual relation associated with the stream. As in the unitemporal stream model, each insert event has a valid interval $[V_s, V_e)$, indicating the application time interval during which the corresponding tuple is in the virtual relation. For any insert event e, $e.V_s < e.V_e$.

The second type of event is a *retraction*, which can shorten the valid interval of a tuple in the virtual relation, by reducing the value of its valid end time from the original V_e value to the V_{newe} value carried by this retraction event. Note that the valid interval of a tuple can be shortened multiple times by a sequence of retraction events in the bitemporal stream (with increasing CEDR time values). In order to associate a retraction event with a tuple in the virtual relation, we require that the retraction event carry the valid interval, as well as the payload content of the tuple it corresponds to in the virtual relation. Therefore, each tuple in the virtual relation is initially created by an insert event, and potentially modified later by 0 or more retraction events. For each tuple in the virtual relation, We refer to this set of events associated with it as an *event chain*.

The third type of event is called a *CTI* (current time increment). A CTI event is similar to a heartbeat or punctuation event in the stream literature [11, 10]. A CTI event with application time t and

$Type$	V_s	V_e	V_{newe}	(P)
Insert	1	∞		P1
CTI	1			
Retraction	1	∞	10	P1
Retraction	1	10	5	P1
Insert	4	9		P2
CTI	10			

Table 1: An Example Bitemporal Stream

V_s	V_e	(P)
1	5	P1
4	9	P2

Table 2: The Canonical History Table Corresponding to Table 1

CEDR time T indicates that the content of the virtual relation up to (application time) t should no longer be changed by any future events in the input stream (i.e., any event whose CEDR time is greater than T).

The schema of a bitemporal stream has the following structure ($Type$, V_s, V_e, V_{newe}; P). $Type$ indicates whether the stream tuple is an insert, retraction, or CTI. V_s and V_e together specify the valid interval of the tuple in the virtual relation. V_{newe} is only specified for a retraction event to indicate the new valid end time of its corresponding tuple. Note that the information of CEDR time is not explicitly represented in the schema. Rather, it is given by the ordering of tuples in the bitemporal stream. P denotes the set of attributes in the payload component of the schema.

We show an example bitemporal stream in Table 1. On this stream, an insert event first occurs with valid interval $[1, \infty)$ and payload P1. Next a CTI event occurs, indicating that the content of the virtual relation up to application time 1 will no longer change. We subsequently see two retraction events on the stream, reducing the valid end time of the same tuple first to 10, and then to 5. Next another tuple is inserted into the virtual relation with valid interval $[4, 9)$, followed by a CTI event with value 10.

For each stream event e, we define its *sync time*, denoted as $sync(e)$, as follows. If e is an insert or a CTI event, $sync(e) = e.V_s$. If e is a retraction, $sync(e) = e.V_{newe}$.

With the notion of sync time, we can formally define out of order events as follows. An event e is *out-of-order*, if there is another event e' such that $e'.CEDR > e.CEDR, and sync(e') < sync(e)$.

The bitemporal stream model generalizes the unitemporal stream model in two aspects. First, tuples can take V_e values other than ∞. This is useful in many application scenarios, for example, when modeling the expiration time of coupons. It is also essential for modeling windows [4]. Also, the ability to shorten valid intervals through retractions enables the stream engine to use fine grained speculative execution [4] as a method of dealing with out of order event delivery [10].

As the bitemporal stream model subsumes the unitemporal stream model, we adopt the bitemporal stream model in this paper. By default, a *stream* refers to a bitemporal stream.

Bitemporal Converter. A bitemporal stream S can be converted into its *canonical history table*, which is a unitemporal stream, by incorporating all the retraction events in S into content changes of the virtual relation. The canonical history table therefore stores the *eventual* content of the virtual relation. Note that while a bitemporal stream is a sequence of events where ordering is important, its canonical history table is a standard relation, where tuple ordering is immaterial. For example, we show in Table 2 the canonical history table of the bitemporal stream in Table 1. The first tuple in Table 2 corresponds to incorporating the two retraction events with $V_s = 1$ in Table 1 into the first insert event in that same table. The second tuple in Table 2 corresponds to the second insert event in Table 1.

Let the function that converts a bitemporal stream to its canonical history table be $\mathcal{S}[\![\cdot]\!]$, referred to as the *bitemporal converter*. Hence, for a stream S, $\mathcal{S}[\![S]\!]$ denotes its canonical history table.

2.2 Valid Streams

A bitemporal stream is a *sequence* of events ordered by the CEDR time attribute. We make the following assumptions on the streams we deal with. First, streams are arbitrarily long but finite sequences of events. Second, we only consider valid bitemporal streams. We say a bitemporal stream is *valid*, if a) the semantics of its CTI events are not violated, b) the CEDR time of any insert event is less than that of any retraction event belonging to the same event chain as the insert, and c) for any two retraction events e_1 and e_2 belonging to the same retraction chain, if $e_1.CEDR < e_2.CEDR$, then $e_1.V_{newe} > e_2.V_{newe}$. Note that we guarantee that all operators produce valid streams when given valid streams. As a result, if all input streams are valid, all internal derived streams and output streams are valid. In addition, there are zero latency, low overhead techniques for converting all streams into valid streams [7]. We therefore assume that these tehcniques are applied automatically whenever streams flow across a communication channel which may introduce arbitrary disorder.

As a stream is a sequence of events, we allow the standard operations defined on sequences to also operate on streams. In particular, we define stream concatenation and stream prefix as follows.

DEFINITION 1. *Given two streams S_1, S_2, the* stream concatenation *$S_1 \cdot S_2$ is defined, only if the resulting sequence is still a valid stream.*

Let c_∞ denote the special CTI event with value ∞. Note that given a stream S, $S \cdot \{c_\infty\}$ is always still valid.

DEFINITION 2. *For two streams S, p, p is a* prefix *of S, denoted as $p \preccurlyeq S$, if the sequence of events corresponding to p is a prefix of the sequence of events corresponding to S.*

We now define the following special classes of streams.

DEFINITION 3. *A stream is* ordered, *if it has no out of order events.*

Intuitively, in an ordered stream, the sync time of its events increase with their CEDR time.

LEMMA 1. *If a stream is ordered, for each insert event in it, there is at most one retraction event belonging to the same event chain as the insert.*

The lemma holds, since in any valid stream, any two retraction events belonging to the same event chain are necessarily out of order.

DEFINITION 4. *A stream is* perfect, *if it is ordered, and has no retraction events.*

LEMMA 2. *Any prefix of an ordered stream is still ordered. Any prefix of a perfect stream is still perfect.*

DEFINITION 5. *A stream S is* closed at *t, if there is a CTI event in S with sync time greater than or equal to t. A stream S is* closed up to *t, if S is closed at t, and not closed at $t + 1$.*

A stream S is closed, *if the last event of S (the event with the largest CEDR time value) is a CTI event, and the application time of this CTI event is larger than that of any other event in S.*

LEMMA 3. *A stream is closed up to t, if and only if the largest sync time of all the CTI events contained in it is t.*

A stream is closed, if and only if it is closed up to t, where t is the sync time of its last CTI event.

LEMMA 4. *If a stream p is closed at t, for any stream S such that $p \preccurlyeq S$, S is closed at t as well.*

If a stream S is closed up to t, there exists a prefix p of S such that p is closed up to t, and is closed.

Note that even if a stream is closed, it may not have any closed prefix.

2.3 Operators and Consistency Levels

For simplicity, we focus on unary operators in the following discussion, but our results can be easily extended to binary operators.

A *logical operator* is a function from canonical history tables to canonical history tables. Recall that operator semantics are defined on canonical history tables. Given a canonical history table H as the input of a logical operator o, $o[\![H]\!]$ denotes the output canonical history table defined by the semantics of o.

A *physical operator* is an algorithm that implements a logical operator, and is a function from streams to streams. Given a stream S as the input of a physical operator f, $f(S)$ denotes the output stream.

Here are some properties that an physical operator may have.

DEFINITION 6. *A physical operator f is* order preserving, *if for any ordered stream S, $f(S)$ is ordered. In addition, $f(S)$ is closed if S is closed.*

A physical operator f is perfectness preserving, *if for any perfect stream S, $f(S)$ is perfect. In addition, $f(S)$ is closed if S is closed.*

DEFINITION 7. *A physical operator f that implements logical operator o is* consistent, *if for any closed stream S, $o[\![\mathcal{S}[\![S]\!]]\!] = \mathcal{S}[\![f(S)]\!]$.*

Intuitively, a physical operator is consistent, if it implements its corresponding logical operator correctly.

In addition to being consistent, A good physical operator should also ensure that it produces output events *incrementally* as it reads events from the input stream, even if the input stream has out of order events. This is important, as the input stream may not be closed, if it contains out of order events. In that case, a physical operator that produces an arbitrary output stream (not conforming the the operator semantics) can still be consistent. This shows that the notion of operator consistency is rather weak.

The incremental output property is captured by the following definition.

DEFINITION 8. *A physical operator f that implements o is* progressive, *if for any stream S closed up to t, $o[\![\mathcal{S}[\![S \upharpoonright t]\!]]\!] = \mathcal{S}[\![f((S \upharpoonright t) \cdot \{c_\infty\})]\!]$.*

Here $S \upharpoonright t$ (*S up to application time t*) consists of the set of events in S whose sync time values are less than t.

The above definition establishes an upper bound for the output latency of a progressive operator. That it, when a progressive operator sees an input CTI event, it must produce enough output events to bring the state of its output stream up to the application time of that input CTI event.

By Lemma 3, we obtain the following result.

LEMMA 5. *Any progressive operator is consistent.*

In the following text, we only consider progressive operators.

In addition to being progressive, a physical operator may also guarantee that its output stream has nice properties such as being ordered or perfect. This is useful to the consumers of its output stream.

DEFINITION 9. *A progressive physical operator f is* ultra consistent, *if for any stream S, $f(S)$ is perfect.*

A progressive physical operator f is strongly consistent, *if for any stream S, $f(S)$ is ordered.*

LEMMA 6. *Any ultra consistent operator is perfectness preserving and is strongly consistent. Any strongly consistent operator is order preserving.*

THEOREM 7. *If operator f is strongly consistent, it can produce output events only on input CTI events.*

PROOF. sketch: Suppose not. We can construct a perfect input stream that causes the operator f to produce retraction events, and therefore produces an output stream that is not perfect. This contradicts the assumption that f is strongly consistent. □

2.4 Output Latency

We would like to formally study the relationship between the output latency of operators and the size of the output streams. We formalize the notion of output latency as follows. Intuitively, given two physical operators f_1 and f_2 implementing the same logical operator o, f_1 has no greater output latency than f_2, if for any input stream S, $\mathcal{S}[\![f_1(S)]\!] \supseteq \mathcal{S}[\![f_2(S)]\!]$. That is, $f_1(S)$ may contain some events whose application time is beyond any event in $f_2(S)$. For example, for a strongly consistent operator f implementing logical operator o, given an input stream S that is closed up to t, f cannot produce output events whose application time is beyond t, as is shown in Theorem 7. However, for another physical operator f' implementing o, where f' is more speculative than f, f' can produce such output events. f' therefore has no greater output latency than f.

DEFINITION 10. *Given a consistent operator f, its* output watermark function τ_f *is a function mapping streams to application timestamps, where $\tau_f(S)$ is defined to be the largest V_e value of all events in $f(S)$.*

For example, for a ultra consistent operator f, let S be closed up to t, then $\tau_f(S) \le t$. This is because any output event with application time beyond t may induce future retraction events in the output stream. A formal proof can be constructed similar to the proof to Theorem 7.

We can compare the output latency of two physical operators implementing the same logical operator as follows.

DEFINITION 11. *For a logical operator o, let f_1 and f_2 be two progressive physical operators implementing o. We say the output latency of f_1 is no greater than that of f_2, if for any input stream S, $\tau_{f_1}(S) \ge \tau_{f_2}(S)$.*

Since a strongly consistent operator only produces output events on input CTI events, its output latency is inversely proportional to the frequency of CTI events in its input stream [2]. Therefore, the output latency of a strongly consistent operator may be large. among all progressive operators implementing the same logical operator, a ultra consistent operator has the largest output latency.

THEOREM 8. *Given a logical operator o, among all progressive physical operators implementing it, the lowest output latency of an order preserving operator is no greater than that of a prefectness preserving operator and that of a strongly consistent operator, which in turn is no greater than that of a ultra consistent operator. The lowest output latency of a prefectness preserving operator and that of a strongly consistent operator are in general not comparable.*

In Section 3, we will present perfectness preserving and order preserving algorithms that achieve lower latency than ultra consistent operators.

On the other hand, the lower the output latency of a progressive operator is, the larger its output stream size is, as is given by the theorem below.

THEOREM 9. *For a logical operator o, let f_1 and f_2 be two progressive physical operators implementing o. If the output latency of f_1 is no greater than that of f_2, for any input stream S, the size of $f_1(S)$ is no smaller than that of $f_2(S)$.*

Putting together the above results, we can conclude that among all progressive physical operators for the same logical operator, the size of the output stream produced by a ultra consistent operator is the smallest. A practitioner concerned with output stream size may therefore favor more conservative physical operators over more speculative ones.

3. ALGORITHMS AND ANALYSIS

In this section, we focus on a general class of unary operators, *snapshot-oriented operators*, and present algorithms implementing them. The semantics of snapshot-oriented operators is introduced in Section 3.1. Section 3.2 describes the data structures used in the algorithms and the invariants that are preserved. Section 3.3 overviews how the correctness proof can be constructed from the invariants, and Section 3.4 presents the algorithms that implement a perfectness preserving operator. Finally, we discuss how the previous algorithms can be easily adapted to implement an ordering preserving operator, and a maximally speculative, true middle consistency operator.

3.1 Semantics of Snapshot Oriented Operators

We formally define the semantics of snapshot oriented operators. This is necessary for the correctness proof of the algorithms implementing the snapshot oriented operators that we will present next.

For a canonical history table H, ENDPT(H) defines an array of distinct end points of the valid intervals of the tuples in H, sorted in an ascending order. Formally, $\text{ENDPT}(H) = sort(\{e.V_s | e \in H\} \cup \{e.V_e | e \in H\})$, where $sort$ takes an input set of values (eliminating duplicates), and produces an output array of these values in an ascending order. The interval between every two consecutive end points in ENDPT(H) is referred to as a *snapshot*.

Given a value $v \in \text{ENDPT}(H)$, $v.next$ is defined to be the smallest value y in ENDPT(H) where $y > v$. Note that if v is the largest in ENDPT(H), $v.next$ is not defined. For technical convenience, we define $\text{ENDPT}^-(H)$ to be ENDPT(H) subtracting the largest value. This way, $v.next$ is defined for any $v \in \text{ENDPT}^- H$.

Given a value $v \in \text{ENDPT}^-(H)$, define F_v to be the set of tuples in H whose valid intervals "cover" the interval defined by t and $v.next$. Formally,

$$F_v = \{e \in H | e.V_s \leq v, e.V_e \geq v.next\}$$

F_v is undefined if $v \notin \text{ENDPT}^-(H)$.

The semantics of a snapshot oriented operator o is as follows.

$$o(H) = \bigcup_{v \in \text{ENDPT}^-(H)} \bigcup_{r \in P_t} (v, v.next; r)$$
$$\text{where } P_v = \bigotimes(F_v)$$

P_v is a set of payload values produced by $\bigotimes$, whose definition is operator specific. For example, if o is the aggregation operator SUM, then $\bigotimes$ is Σ. Intuitively, in this case, for each set of tuples defined by F_v, the SUM operator aggregates over them on a specified aggregation attribute, and for each value $v \in \text{ENDPT}(H)$ except for the last one, produces exactly one output tuple whose valid interval is defined by v and $v.next$.

3.2 Data Structures and Invariants

The algorithms for snapshot oriented operators are organized into three event processing algorithms. These three algorithms correspond to the actions taken when receiving each of the three types of events (inserts, retractions, and CTIs).

The insert and retraction algorithms are organized into three phases: The first phase, called the retraction phase, issues retractions for previously output events which are impacted by the incoming event. The second phase, called the snapshot update phase, updates the internal operator state impacted by the incoming event. The third phase, called the issue phase, issues the necessary insert events. CTIs have an additional fourth phase, called the cleanup phase.

The first presented algorithms have minimum latency amongst all algorithms which guarantee that perfect input produces perfect output.

We begin with some terminology and a description of the data structures used throughout the algorithms. In the following text, S denotes the input stream, and O denotes the output stream produced by the operator algorithms reading S as input.

Output watermark τ. τ is a point application timestamp, representing the largest V_e value that has appeared in the output stream events so far. As it will become clear later, it corresponds to values produced by the output watermark function defined in Section 2.4.

Invariant τ-O: For each $e \in \mathcal{S}[\![O]\!]$, $e.V_e \leq \tau$. Also, if S is perfect, exactly all tuples in $\mathcal{S}[\![O]\!]$ with $V_e \leq \tau$ have been produced by the events in O, and O is perfect.

Invariant τ-S-Algorithms: Let S be closed up to t. After processing an input event e in S,

$$\tau := \begin{cases} \max(\tau, e.V_s) & \text{if } e \text{ is an insert} \\ \tau & \text{if } e \text{ is a retraction} \\ \max(\tau, \tau') & \text{if } e \text{ is a CTI} \end{cases}$$

Here τ' the largest value in $\text{ENDPT}(\mathcal{S}[\![S]\!])$ where $\tau' \leq e.V_s$.

Invariant τ-S-Algorithms shows that τ monotonically increases with the prefixes of S that have been processed.

THEOREM 10. *Given a snapshot oriented operator o, among all perfectness preserving operators implementing o, the operator whose output watermark function is defined by Invariant τ-S-Algorithms has the lowest output latency.*

The proof is similar to the one to Theorem 7. Essentially, we can show that for a perfectness preserving operator f, if for some input insert event e in a perfect input stream S, the new value of τ is greater than $\max(\tau, e.V_s)$ as is defined by Invariant τ-S-Algorithms, then we can construct another perfect input stream S'

by extending S, such that f has to produce retraction events in order to maintain its correctness on processing S', thus violating the assumption that f is perfectness preserving.

POS. The discussion of the algorithms below is organized around an internal structure, called the *Previous Output Synopsis*, or *POS*. The POS is an associative data structure, keyed on an application point timestamp *TS*, and valued on an abstract data type *State*, of which the set of allowed operations will be described below. The entries in POS are organized into an efficiently searchable access structure ordered on TS (e.g. a B+ tree). It is guaranteed that there are no duplicate TS entries in POS.

Each POS entry is a key, value pair, denoted as (TS, State). The following operations are defined on POS.

- *POS.First* denotes the entry in POS with the smallest TS value.
- *POS.Last* denotes the entry in POS with the largest TS value.
- Given a timestamp T, *POS.SearchLE(T)* finds the entry (TS, State) in POS with maximal TS value where TS $\leq$ T.
- *POS.Search(V_s, V_e)* returns all entries in POS which correspond to output events with valid intervals overlapping with $[V_s, V_e)$. The returned entries are sorted in the ascending order of TS. More precisely, the first returned entry is the entry with maximal TS such that TS $\leq V_s$. The last returned entry is the POS entry with maximal TS such that TS $< V_e$.
- *POS.ISearch(V_s, V_e)* is similar to POS.Search(V_s, V_e), with a slightly different search criteria. It returns all entries in POS which correspond to output overlapping with $[V_s, V_e]$. More precisely, the first returned entry is the entry with maximal TS such that TS $\leq V_s$. The last returned entry is the POS entry with maximal TS such that TS $\leq V_e$.
- *POS.SearchEQ(T)* returns a POS entry p with p.TS=T. Such a POS entry must exist; otherwise the behavior is undefined.
- *POS.Insert(TS, State)* adds a new entry (TS,State) to POS.
- *POS.Remove(T)* removes the entry keyed on T. This entry must exist.

For a POS entry p, *p.next* denotes the POS entry with the smallest TS value among those entries with TS value greater than p.TS. If p is POS.Last, p.next is undefined. Dually,*p.prev* denotes the POS entry with the largest TS value among those entries with TS value less than p.TS. If p is POS.First, p.prev is undefined.

For the abstract data type *State*, we define the following operations. Assume p is an arbitrary POS entry.

- *p.State.GetPayloads* constructs and returns the set of output payloads associated with p.
- *p.State.Add(P)* incorporates the P into the State associated with p.
- *p.State.Remove(P)* removes the payload P from the State associated with p.
- *p.State.k* returns the number of events which "contribute" to the value of p.State. Specifically, it starts at 0, is incremented for each call to State.Add, and decremented for each call to State.Remove. When this number drops down to 0 again, the entry p is automatically removed from POS.

Note that the semantics of p.Sate.GetPayloads, p.State.Add(P) and p.State.Remove(P) are operator specific. For example, if the snapshot oriented operator is the COUNT aggregate, p.State simply maintains a count value, to be returned as a singleton set at the call to p.Sate.GetPayloads. That count value is incremented at each call to p.State.Add(P), ignoring the content of P, and similarly decremented at each call to p.State.Remove(P). However, for some other snapshot oriented operators such as Rank, p.State needs to store the set of payloads that have been added to p.State via the calls to p.State.Add, and that have not been removed via the calls to p.State.Remove. In general, for a POS entry p, if p.TS = v for some $v \in \mathrm{ENDPT}^-(\mathcal{S}[\![S]\!])$. p.State.GetPayloads produces the set of payload values P_v defined in Section 3.1.

Invariant POS-τ-S-O: Let S be closed up to t. For each $v \in \mathrm{ENDPT}(\mathcal{S}[\![S]\!])$: the following conditions hold.

- If $v < \tau$,[1] there is a one-to-one correspondence between payload values in P_v and the set of tuples in $\mathcal{S}[\![O]\!]$ with valid interval defined by $[v, v.next)$. If there exists entry p in POS such that p.TS=v, p.State.GetPayloads=P_v.
- If $v = \tau$, POS.Last.TS = v. Furthermore, the set of payload values associated with POS.Last.State is $\{e.P : e \in ECQ\}$
- If $v \in \mathrm{ENDPT}^-(\mathcal{S}[\![S]\!]), v < t$ and $v.next < t$, there is no entry in POS with TS$\leq v$. That is, the state stored in POS is purged as aggressively as possible on input CTI events from S.
- If $v \in \mathrm{ENDPT}^-(\mathcal{S}[\![S]\!]), v < t$ and $v.next \geq t$, POS.First.TS=v.

Note that POS initially has one entry ($-\infty$, EmptyState). Here EmptyState corresponds to the value for State which signifies that there are no entries in the snapshot (see discussion of Sum for an example). Also, the initial value of τ is $-\infty$.

ECQ. In order to also guarantee provably good performance in the worst case, we need a priority queue data structure, called the *Endpoint Compensation Queue*, or *ECQ*. It is keyed on *TS*, and valued on *Tuple*, the abstract data type representing a tuple in canonical history tables. Recall from Section 2 that a Tuple has schema (V_s, V_e; P). Similar to the entries in POS, the entries in ECQ are sorted on TS. Different from POS, ECQ can have multiple entries with the same TS value.

Each ECQ entry is a key, value pair, denoted as (TS, Tuple). ECQ maintains the invariant that for each entry (T, e), T = $e.V_e$.

The following operations are defined on ECQ.

- *ECQ.Empty* returns True if the queue is empty. Otherwise it returns false.
- *ECQ.Head* returns the first entry (i.e., an entry with the smallest TS value) in ECQ. It is undefined if the queue is empty.
- *ECQ.Pop* returns and removes the head element in ECQ. It is undefined if the queue is empty.
- *ECQ.Insert(Tuple)* inserts a new entry (Tuple.V_e, Tuple) into ECQ.
- *ECQ.Remove(Tuple)* removes an existing entry (Tuple.V_e, Tuple) from ECQ. This entry must exist.

Invariant ECQ-τ-S: The set of tuples contained in ECQ is $\{e \in \mathcal{S}[\![S]\!] : e.V_e > \tau\}$.

[1]In this case, since we know $\tau \in \mathrm{ENDPT}(\mathcal{S}[\![S]\!])$ by invariant τ-O, $v \in \mathrm{ENDPT}^-(\mathcal{S}[\![S]\!])$ must hold true.

3.3 Overview of Correctness Proof

Let $\mathcal{I}$ be the set of invariants containing τ-O, τ-S-Algorithms, POS-τ-S-O and ECQ-τ-S, all of which are described in Section 3.2.

Given an input stream S, we will prove that the algorithms to be presented below uphold all the invariants in $\mathcal{I}$. This is done by induction on the prefixes of S. Specifically, we will show that in the base case, the initial values of τ, POS, ECQ, S and O satisfy the invariants. In the inductive case, we will show that if the invariants hold for some $p \preccurlyeq S$, after processing the event in S following p, the invariants still hold for the values τ, POS, ECQ and O updated by the algorithms. The proofs to these claims can be found in the appendix.

Once we show that the algorithms always uphold the invariants, the correctness of the algorithms follow straightforwardly.

THEOREM 11. *If the algorithms uphold all the invariants in $\mathcal{I}$, then the physical operator f that they together implement is consistent, and perfectness preserving.*

PROOF. Given any closed stream S. Note that $O = f(S)$ by definition. Let the logical operator implemented by f be o. By the correspondence between P_v and $\mathcal{S}[\![O]\!]$ in the first bullet of Invariant POS-τ-S-O, $\mathcal{S}[\![O]\!] = o[\![\mathcal{S}[\![S]\!]]\!]$. Therefore, f is consistent.

Invariant τ-O shows that f is perfectness preserving. □

3.4 Algorithms

In order to indicate when and what output is produced in the following algorithms, we introduce a few simple output producing functions. *OutputInsert(V_s, V_e, P)* produces an insert event e with $e.V_s = V_s, e.V_e = V_e, e.P = P$. *OutputRetraction($V_s$, V_e, V_{newe}, P)* produces a retraction event e with $e.V_s = V_s, e.V_e = V_e, e.V_{newe} = V_{newe}, e.P = P$. *OutputCTI($V_s$)* produces a CTI event e with $e.V_s = V_s$.

We are now ready to give the algorithms for our snapshot-oriented operators.

3.4.1 Handling Insert Events

Phase 1. We must first identify all the previously output events which must be retracted. To do so, we first invoke POS.Search to obtain a set of POS entries affected by the input insert event. From each such entry, and the TS of the next entry, we construct the correct retractions. Note that end points must be handled with care. This results in the following algorithm for phase 1 of insert (Line 1 – Line 7 in Algorithm 1).

Complexity analysis of Phase 1. In our complexity analysis, we state complexity in terms of n =#input events, and m =#output events. This allows us to establish and compare ourselves to the lower bound on optimality: $O(n + m)$. Note that for each insert, the amount of work done in phase 1 is $O(\log n+m+1)$ in the worst case. The log factor comes from the B+ tree lookup to retrieve the first retracted entry, while m corresponds to the number of POS entries which are retrieved and retracted. The 1 comes from the worst case assumption that we always retrieve the last entry of POS, which is never retracted since it never produced prior output. The complexity of phase 1, given n inputs, is therefore $O(n * (logn + 1) + m) = O(nlogn + m)$

Note that we do not multiply m by n since the sum of all output over all inserts is m.

Phase 2. Note that for convenience, given an event e, we define *Tuple(e)* := the tuple ($e.V_s$,$e.V_e$, $e.P$). The algorithm for Phase 2 is shown between Line 8 and Line 19 in Algorithm 1.

In the while loop, we create entries in POS from the unprocessed endpoints in ECQ which are behind $V_s$of the incoming event. This

Algorithm 1 Algorithm for Insert Events

```
Require: Input insert event $e$
    // phase 1
 1: for all entry $p$ in POS.Search($e.V_s$, $e.V_e$) ordered by TS do
 2:     if $p$ == POS.Last then
 3:         break
 4:     OutputPayloads := $p$.State.GetPayloads
 5:     OutTS := $\max(e.V_s, p.\text{TS})$;
 6:     for all payload $x$ in OutputPayloads do
 7:         OutputRetraction($p$.TS, $p$.next.TS, OutTS, $x$)

    // phase 2
    // Maintain ECQ
 8: Invoke Algorithm 2 with $V_s = e.V_s$
    // Insert new entries into POS due to $e$
 9: ($t_2$, State$_1$) := POS.SearchLE($e.V_s$)
10: if $t_2 < e.V_s$ then
11:     POS.Insert($e.V_s$, copy of State$_1$)
12: if $e.V_e \leq \tau$ then
13:     ($t_3$, State$_2$) := POS.SearchLE($e.V_e$)
14:     if $t_3 < e.V_e$ then
15:         POS.Insert($e.V_e$, copy of State$_2$)
16: else // Maintain invariant ECQ-$\tau$-$S$
17:     ECQ.Insert(Tuple($e$))
    // Incorporate $e$ into the relevant POS entries
18: for all entry $p$ in POS.Search($e.V_s, e.V_e$) ordered by TS do
19:     $p$.State.Add($e.P$)

    // phase 3
20: if $\tau < e.V_s$ then // $e$ increases the output watermark $\tau$
21:     Invoke Algorithm 3 with $V_s = \tau, V_e = e.V_s$
22:     $\tau := e.V_s$
23: else // output events whose valid intervals intersect that of $e$
24:     for all entry $p$ in POS.ISearch($e.V_s, e.V_e$) ordered by TS do
25:         if ($p$ == POS.Last) || ($p$.TS = $e.V_e$ && $p$.State.k > 1) then
26:             break
27:         OutputPayloads := $p$.State.GetPayloads
28:         for all payload $x$ in OutputPayloads do
29:             OutputInsert($p$.TS, $p$.next.TS, $x$)
```

Algorithm 2 Algorithm for Moving Data from ECQ to POS

```
Require: A timestamp $V_s$
 1: while !ECQ.empty && ECQ.Head.TS $\leq V_s$ do
 2:     $(t_1, e')$ := ECQ.Pop
 3:     NewState := copy of POS.Last.State
 4:     NewState.Remove($e'.P$)
 5:     POS.Insert($t_1$, NewState)
```

Algorithm 3 Algorithm for Producing Insert Events from POS entries

```
Require: Two timestamps $V_s$, $V_e$ where $V_s < V_e$
 1: for all entry $p$ in POS.Search($V_s$, $V_e$) ordered by TS do
 2:     if $p$ == POS.Last then
 3:         break
 4:     OutputPayloads := $p$.State.GetPayloads
 5:     for all payload $x$ in OutputPayloads do
 6:         OutputInsert($p$.TS, $p$.next.TS, $x$)
```

way, we maintain our post-conditions for both ECQ and POS. We then make sure that we have appropriate entries in POS for $e.V_s$, and $e.V_e$if $e.V_e$is behind τ. This also is done to maintain our post-conditions on POS. If $e.V_e$ is after τ, we instead add an entry to ECQ, which ensures that ECQ conforms to our post-conditions. Now that we have all required entries in POS, we use the for loop to add the event to the state associated with all affected entries in POS. After phase 2, all post-conditions on POS and ECQ are upheld.

Complexity analysis of Phase 2. In terms of complexity, since every input event's endpoint may be added and removed only once in this structure over the lifetime of the operator, and since the total number of checks of the head cannot exceed the number of input events, the total complexity of this loop is $O(n \log n + n) = O(n \log n)$

For each call to Insert, the if conditions which potentially add new entries to POS, have complexity $O(1)$. The final for loop is guaranteed to update at most $m + 1$ intervals. This is the result of our post-condition that all but the last entry in POS must be updated in the output. The complexity of this operation is therefore $O(m + n)$ over all inserts. The total complexity of phase 2 is therefore $O(n \log n + m + n) = O(n \log n + m)$.

Phase 3. Phase 3 must ensure that all necessary inserts are output. The algorithm is shown between Line 20 and Line 29 in Algorithm 1.

Phase 3 is divided into 2 cases. In the first case, $\tau < e.V_s$. In this case, the output must "catch up" to the new τ. We must therefore produce output for all POS entries between the old τ and the new τ. We therefore first search the POS and then generate all necessary output. We then adjust τ to maintain our post-condition. This case has worst case complexity $O(\log n + m + 1)$ per insert call, or $O(n \log n + m + n) = O(n \log n + m)$ over all inserts if it is always chosen.

The second case is where $e.V_s \leq \tau$. In this case, we simply need to produce output for all POS entries which were previously corrected during phase 2, except for possibly the last POS entry, which may have been corrected, but cannot be output. It is easy to see that the worst case complexity, like the previous case, is $O(\log n + m + 1)$ per insert call, and therefore O($n \log n + m$) over all inserts if it is always chosen.

Complexity analysis of Phase 3. Since the complexity is the same in both cases, the complexity of phase 3 is $O(n \log n + m)$.

We may now determine the overall complexity of insert, which is $O(n \log n+m)+O(n \log n+m)+O(n \log n+m) = O(n \log n+m)$.

3.4.2 Handling Retraction Events

Phase 1. The phase 1 algorithm for retraction is very similar to the phase 1 algorithm for insert. The difference is that instead of generating retractions for the interval [V_s, V_e), we use the interval [V_{newe}, V_e], and issue a retraction on the POS entry with TS=$V_e$iff we are going to remove the POS entry later. The result is shown between Line 1 and Line 13 in Algorithm 4.

Using reasoning identical to insert, the complexity is $O(n \log n + m)$.

Phase 2. Phase 2, unlike phase 1, is different than the corresponding algorithm for insert. This algorithm, in addition to potentially creating a new entry in POS, may also remove an entry from POS, and potentially add and/or remove an entry from ECQ.

Note that for convenience, given a retraction event e, we define *OldTuple(e)* := the tuple ($e.V_s$,$e.V_e$, $e.P$), and *NewTuple(e)* := the tuple ($e.V_s$, $e.V_{newe}$, $e.P$). Note that both of these function can be computed in constant time.

The algorithm is shown between Line 14 and Line 27 in Algo-

Algorithm 4 Algorithm for Retraction Events

```
Require: Input retraction event $e$
    // phase 1
 1: for all entry $p$ in POS.ISearch($e.V_{newe}, e.V_e$) ordered by TS do
 2:   if ($p$ == POS.Last) || ($p$.TS = $e.V_e$ && $p$.State.k > 1) then
 3:     break
 4:   OutputPayloads := $p$.State.GetPayloads
 5:   OutTS := max($e.V_{newe}$, $p$.TS);
 6:   for all payload $x$ in OutputPayloads do
 7:     OutputRetraction($p$.TS, $p$.next.TS, OutTS, $x$)
 8: if $e.V_{newe} == e.V_s$ then // $e$ is a full retraction
 9:   $p'$ = POS.SearchEQ($e.V_s$)
10:   if $p'$.State.k == 1 then // need to issue full retractions
11:     OutputPayloads$'$ := $p'$.State.GetPayloads
12:     for all payload $y$ in OutputPayloads$'$ do
13:       OutputRetraction($p'$.prev.TS, $p'$.TS, $p'$.prev.TS, $y$)

    // phase 2
    // Maintain ECQ
14: if $e.V_e > \tau$ then
15:   ECQ.Remove(OldTuple($e$))
16: if $e.V_{newe} > \tau$ then
17:   ECQ.Insert(NewTuple($e$))
    // Maintain POS
18: if $e.V_e \leq \tau$ then
19:   ($t_1$, State$_1$) := POS.SearchEQ($e.V_e$)
20:   if State.k = 1 then
21:     POS.Remove($e.V_e$)
22: if $e.V_{newe} \leq \tau$ then
23:   ($t_2$, State$_2$) := POS.SearchEQ($e.V_{newe}$)
24:   if $t_2 < e.V_{newe}$ then
25:     POS.Insert($e.V_{newe}$, copy of State$_2$)
26: for all entry $p$ in POS.Search($e.V_{newe}, e.V_e$) ordered by TS do
27:   $p$.State.Remove($e.P$)

    // phase 3
28: Invoke Algorithm 3 with $V_s = e.V_{newe}, V_e = e.V_e$
```

Algorithm 5 Algorithm for CTI Events

```
Require: Input CTI event $e$
    // phase 1: no operation

    // phase 2: Maintain ECQ and POS
 1: Invoke Algorithm 2 with $V_s = e.V_s$

    // phase 3
 2: if $\tau < e.V_s$ then
 3:   Invoke Algorithm 3 with $V_s = \tau, V_e = e.V_s$
 4:   $\tau$ := POS.Last.TS
 5: OutputCTI($e.V_s$)

    // phase 4
 6: while (POS.First != POS.Last) && (POS.First.next.TS < $e.V_s$) do
 7:   POS.Remove(POS.First.TS)
```

rithm 4.

The first two if statements adjust the endpoints of the retraction interval so that all structures contain appropriate entries. This reflects the potential removal of entries associated with $e.V_e$and the potential introduction of endpoints associated with $e.V_{newe}$. The most complex operations are Find, and SearchLE, both of which are $O(\log n)$. The for loop makes any correction to State in affected POS entries. Like insert, this loop is guaranteed to update at most m+1 intervals. As a result, similar to insert, the complexity of phase 2 is $O(n \log n + m)$.

Phase 3. Like Phase 1, Phase 3 is somewhat different from Phase 3 of insert, and is actually simpler. For instance, it is not possible that τ has changed. Also, we are outputting using the interval $[e.V_{newe}, e.V_e)$. Finally, the end of the insertion interval is handled more simply. This results in Line 28 of Algorithm 4.

It is easy to see, given our previous analysis of other phases, that this is also $O(n \log n + m)$.

The worst case complexity of retraction is, like insert, $O(n \log n + m)$.

3.4.3 Handling CTI Events

Unlike the other event types, the algorithm for CTI is broken into 4 phases. While the first 3 phases correspond to the same phases for other event types, the last phase is unique to CTI. It is the cleanup phase, and is the only place amongst all algorithms where state may be freed.

Phase 1. Since no retractions may ever be issued as a result of CTIs, the first phase is empty.

Phase 2. In this phase, we may need to prepare to produce any output associated with entries (TS, Event) in the ECQ with TS < $e.V_s$. The algorithm for phase 2 is identical to the beginning of phase 2 for insert and is shown in Line 1 in Algorithm 5.

As in insert, the complexity of this loop is $O(n \log n)$.

Phase 3. Similar to phase 2, phase 3 is very similar to phase 3 for insert. The main difference is that we only want to produce output if the CTI moves the output time forward. Also, we generate an output CTI. The algorithm is shown between Line 2 and Line 5 in Algorithm 5.

Using similar reasoning to previous phases, the complexity of this phase is: $O(n \log n + m)$.

Phase 4. In this phase, we release the memory associated with output events which are no longer volatile. For POS, this is all the pairs p is an element of POS where p.next < $e.V_s$. Since ECQ only contains information about future output, we are not able to perform any cleanup. The resulting algorithm is shown between Line 6 and Line 7 in Algorithm 5.

In terms of complexity, the number of times the loop condition is checked is the number of CTI events received, plus the number of entries ever added to the POS. Since the maximum number of entries ever added to the POS is O(n), the complexity of this phase is: O(n)

The overall complexity of CTI is therefore also $O(n \log n + m)$.

3.5 Other Levels of Latency

In the previous section, we developed minimally latent algorithms which guaranteed that perfect input produced perfect output. In this section, we outline the changes necessary to implement operators with lower latency and weaker guarantees. Since these algorithms are highly related to the given algorithms, we simply describe the changes informally.

Order preserving. Here we describe the minimally latent algorithms which guarantee that ordered input produces ordered output. In practice this means that the algorithms may produce one retraction per event in the canonical history table, and we must use this flexibility to maximum advantage in terms of latency for ordered input. This is a very simple change from the algorithms in Section 3.4. Instead of not producing output for the last entry in POS, we produce output events e with $e.V_e = \infty$, and retract the output when a new entry is appended to POS. There are no changes to the way in which internal structures are maintained.

For instance, at the end of the first phase of insert, we check if $e.V_s > \tau$. If it is, we retract the output associated with the last entry of POS from infinity to $e.V_s$. In the third phase of insert, before "τ = $e.V_s$", we output inserts for POS.Last. These inserts have $V_e = \infty$. Changes similar in scope are necessary for handling retractions and CTIs.

Note that this increases the complexity of each algorithm no more than a constant factor, and therefore has no impact on asymptotic complexity.

Minimum latency (true middle consistency). In the algorithm which minimizes latency, no ECQ is necessary since, after any event is processed, every event endpoint in the current input canonical history table must have corresponding output in the current output canonical history table. As a result, every volatile event endpoint in the current input canonical history table has an associated POS entry. Also, output must be produced for each one of these entries. While much more output may be produced, and m, as a result, may be much higher, asymptotic efficiency in terms of n and m is unchanged from $O(n \log n + m)$.

4. RELATED WORK

While there is little work on out of order delivery of stream data, the work on stream buffering [10] and punctuations [11] is perhaps the most useful. There, the authors suggest buffering and waiting for an external guarantee that all data has been delivered up to some application time before allowing the data to be processed by the system. This allows unary operators to assume that all data arrives in order, and allows n-ary ($n > 1$) operators to assume that each individual input stream arrives in order. This implements one extreme point (highest level of consistency) on our consistency spectrum. More recently, [2] studied the trade-offs between output latency and memory usage with different frequencies of punctuations in data streams.

Closer in spirit, although not in fact, is the work presented in [3], which proposes the use of a rewind event to implement a very coarse form of speculative output. Specifically, [3] proposes this as a mechanism for handling the exceptional case of an input source in a distributed system going down for a long period of time. However, the coarseness of the retraction makes the approach impractical for active use, and the resulting algorithms and design decisions are not formally motivated, described, or analyzed.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the first algorithms which implement the consistency spectrum presented in [4]. More specifically, we have formally identified interesting new design considerations in the construction of such algorithms, and given stream processing algorithms for a sophsiticated operator, Aggregate. We have also provided an asymptotic analysis which shows that these algorithms are $O(n \log n + m)$, where n is input size, and m is output size. The algorithms are, therefore, close to optimal ($O(n + m)$).

The algorithms presented here represent a viable starting point for speculative streaming algorithms, and are an important initial step in the implementation of such a system.

While this work establishes viable operator algorithms for one of the more challenging CEDR operators, future work includes developing the necessary algorithms for other operators, and establishing either slightly tighter lower bounds on optimality, or slightly less asymptotically expensive algorithms. Also, while the algorithm properties described in this paper are interesting, there may be other interesting desirable properties, leading to other interesting design points.

6. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[2] Y. Bai, H. Thakkar, H. Wang, and C. Zaniolo. Optimizing timestamp management in data stream management systems. In *Proc. ICDE*, 2007.

[3] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. SIGMOD*, 2005.

[4] Roger Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *Proc. CIDR*, 2007.

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

[6] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.

[7] J. Goldstein and M. Hong. Consistency sensitive operators in cedr. 2007. TR ID MSR-TR-2007-158.

[8] C. S. Jensen and R. T. Snodgrass. Semantics of time-varying attributes and their use for temporal database design. In *Fourteenth International Conference on Object-Oriented and Entity Relationship Modeling*, 1995.

[9] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, 2003.

[10] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. PODS*, pages 263–274, 2004.

[11] P. Tucker and D. Maier. Exploiting punctuation semantics in data streams. In *Proc. ICDE*, page 279, 2002.

APPENDIX

Recall that the set of invariants $\mathcal{I}$ is defined in Section 3.3.

LEMMA 12. *The initial values of τ, POS, ECQ, S and O satisfy the invariants in $\mathcal{I}$.*

PROOF. Recall that POS initially contains one entry ($-\infty$, EmptyState), and τ is initialized to $-\infty$. Also, ECQ, S and O are $\emptyset$ initially. The invariants hold trivially in this case as follows.

τ-O holds, as O is empty. τ-S-Algorithms is defined only inductively, without specifying the initial value of τ. So it holds for any initial value of τ. ECQ-τ-S and POS-τ-S-O hold, as S is empty. In addition, note that even if there is no v in $\mathrm{ENDPT}(\mathcal{S}[\![S]\!])$, POS.Last.TS=$\tau$=$-\infty$ in this case (the second bullet of POS-τ-S-O).

□

LEMMA 13. *Algorithm 1 upholds the set of invariants in $\mathcal{I}$.*

PROOF. For an input insert event e, let τ_{old} and τ_{new} respectively denote the values of variable τ before and after e is processed. Similarly, let S_{old} and S_{new} respectively denote the value of the input stream S before and after e is processed. Clearly, $S_{new} = S_{old} \cdot \{e\}$. Let O_{old} and O_{new} respectively denote the value of the input stream O before and after e is processed. Clearly, $O_{old} \preccurlyeq O_{new}$.

We want to prove if all the invariants are satisfied with respect to the values $\tau_{old}, S_{old}, O_{old}$, and the content of POS and ECQ before e is processed, then they are still satisfied with respect to the values $\tau_{new}, S_{new}, O_{new}$, and the content of POS and ECQ after e is processed.

Invariant τ-S-Algorithms: In Algorithm 1, the value of variable τ either remains unchanged, or is updated to $e.V_s$ in Line 22, when $\tau_{old} < e.V_s$. Therefore, $\tau_{new} = \max(\tau_{old}, e.V_s)$. So Invariant τ-S-Algorithms is satisfied.

Invariant ECQ-τ-S: ECQ is possibly updated in Line 8 and Line 17 of Algorithm 1.

If $e.V_e \leq \tau_{old}$, then $e.V_s < e.V_e \leq \tau_{old}$. By assumption, any entry in ECQ has TS value greater than τ_{old}, and therefore is greater than $e.V_s$. Therefore, the invocation of Algorithm 2 in Line 8 has no effect on ECQ. Also, Line 17 is not executed. Therefore the content of ECQ remains unchanged. Since $\tau_{old} = \tau_{new}$ in this case, Invariant ECQ-τ-S remains satisfied.

Otherwise, $e.V_e > \tau_{old}$. In Line 17 of Algorithm 1, e is inserted into ECQ. Since $e.V_e > e.V_s$, we know $e.V_e > \tau_{new} = e.V_s$. Therefore, according to Invariant ECQ-τ-S, e should be in ECQ after e is processed. Also, if $\tau_{new} = e.V_s > \tau_{old}$, according to ECQ-$\tau$-$S$, all the ECQ entries with TS no greater than τ_{new} should be removed. This is realized by the invocation of Algorithm 2 in Line 8. Therefore, Invariant ECQ-τ-S is satisfied in all cases.

Invariant POS-τ-S-O: there are four bullets in this invariant, which we refer to respectively as B1 through B4.

Let S_{old} be closed up to t, note that S_{new} is still closed up to t. By the semantics of CTI events and the definition of t, $e.V_s \geq t$.

Let $v \in \mathrm{ENDPT}(\mathcal{S}[\![S_{new}]\!])$. We first prove B3. The only potential entries to be inserted into POS are the ones with TS values being $e.V_s$ or $e.V_e$ (Line 11 and 15). Specifically, regardless of whether Line 11 is executed, it is guaranteed POS after processing e contains an entry with TS=$e.V_s$. The POS entry with TS=$e.V_e$, inserted into POS by Line 15, is conditioned on $e.V_e \leq \tau$.

So if $v < t, v.next < t$, then $v < e.V_s, v.next < e.V_s$, and there is still no POS entry with TS $\leq v$. So B3 remains satisfied.

Next we prove B4. If $v \leq t, v.next > t$ in POS before processing e, by assumption POS.First.TS=v before processing e. that entry remains in POS after processing e, and the potential new POS entries with TS values being $e.V_s$ or $e.V_e$ are guaranteed to be ordered after that entry with TS=v. Therefore, in POS after processing e, POS.First.TS=v. So B4 is satisfied.

The proof to B2 is split into two cases. In case 1, $\tau_{old} \geq e.V_s$. So $\tau_{new} = \tau_{old}$ in this case. If $e.V_e > \tau_{old}$, Line 15 is not executed, and so the last entry in POS still has TS=τ_{old}. Also, by Line 19, $e.P$ is added to the last entry in POS. Also, e is also added to ECQ in this case, B2 remains satisfied. On the other hand, if $e.V_e \leq \tau_{old}$, the POS entry with TS=$e.V_e$ is inserted by Line 15, but the last entry in POS still has TS=τ_{old}. Also, $e.P$ is not added to that last entry, as the last POS entry will not be returned by POS.Search($e.V_s, e.V_e$) executed in Line 18 in this case. As we have argued above, e is not added to ECQ in this case. Again B2 remains satisfied.

In Case 2, $\tau_{old} < e.V_s$. Then $\tau_{new} = e.V_s$. So $e.V_e > e.V_s = \tau_{new} > \tau_{old}$. So Line 15 is not executed in this case, an the last entry of POS after processing e has TS=$e.V_s$. Also, that last entry is a newly inserted entry by Line 11. Since its state is copied from the previously last entry in POS, by assumption, the set of tuples corresponding to it coincide with the tuples in ECQ before processing e. Since e is added to both ECQ and the new last entry in POS in this case, B2 remains satisfied.

Finally, the proof to B1 is presented as follows. If $\tau_{old} < e.V_s$, POS.Search invoked in Line 1 returns only the last entry of POS. As a result, Line 7 is never executed, so no retraction event is produced in the output stream. Therefore, for any $v \in \mathrm{ENDPT}(\mathcal{S}[\![S_{new}]\!])$, if $v < \tau_{old}$, B1 still holds for v. Next, Line 21 is executed in this case. As a result, for any $\tau_{old} \leq v < e.V_s = \tau_{new}$, there is a POS entry p with TS=v after processing e, and p.State.GetPayloads = P_v where P_v is defined on S_{new}. Furthermore, for each such v, the tuples in $\mathcal{S}[\![O_{new}]\!]$ with valid interval $[v, v.next)$ are produced from the payload values in p.State.GetPayloads. Therefore, there is a one-to-one correspondence between payload values in P_v and this set of tuples in $\mathcal{S}[\![O_{new}]\!]$. Therefore, B1 holds in this case.

If $e.V_s \leq \tau_{old} < e.V_e$, the tuples between $e.V_s$ and τ_{old} are retracted in $\mathcal{S}[\![O_{new}]\!]$ in Line 7. Also, the POS entries between $e.V_s$ and τ_{old} are updated to incorporate e in Line 19. These POS entries are then used to produce new tuples between $e.V_s$ and τ_{old} in $\mathcal{S}[\![O_{new}]\!]$ in Line 29. Therefore, by a similar argument to the above case, B1 again holds in this case.

In the last case, if $\tau_{old} \geq e.V_e$, the argument is similar to the previous case, except that the interval during which events are retracted and later inserted again is between $e.V_s$ and $e.V_e$, instead of between $e.V_s$ and τ_{old}. We note the "if condition" in Line 25 in this case. If the POS entry p is the last entry, by the invariants, we never produce any output event for that entry. Also, if p.TS=$e.V_e$, it may correspond to a newly inserted POS entry by Line 15, where p.State.k = 1, and therefore should be used to produce a new insert event. However, if p.State.k > 1, this means the POS entry with TS=$e.V_e$ already exists before processing e, and Line 15 is not executed. In this case, there is already a tuple in $\mathcal{S}[\![O_{old}]\!]$ corresponding to that POS entry, which should remain in $\mathcal{S}[\![O_{new}]\!]$ as its value is not changed. We therefore do not produce a new insert event in this case.

Invariant τ-O: Given that we have proved POS-τ-S-O, τ-O follows straightforwardly. First, by B2 in POS-τ-S-O, for any POS entry p, p.TS $\leq \tau_{new}$. When processing e, the potential new output events produced by Algorithm 1 all have their $V_e$values set to the TS value of some POS entry (Line 7, 21 and 29). Therefore, for each $e \in \mathcal{S}[\![O_{new}]\!]$, $e.V_e \leq \tau_{new}$. Also, if S is perfect, for each insert event e, $e.V_s \geq \tau_{old}$. In this case, on input e, Algorithm 1 produces a set of new insert events whose valid intervals lie between τ_{old} and $e.V_s$ (Line 21). The output stream produced this way is clearly perfect as well.

□

LEMMA 14. *Algorithm 4 upholds the set of invariants in $\mathcal{I}$.*

PROOF. Invariant τ-S-Algorithms: Clearly, Algorithm 4 does not update τ. Therefore this invariant holds.

Invariant ECQ-τ-S: Line 14 through Line 17 ensures that: if $e.V_e > \tau$, e is removed from ECQ; if $e.V_{newe} > \tau$, e is added back to ECQ with the key value TS set to $e.V_{newe}$. Therefore, this invariant holds.

Invariant POS-τ-S-O: There are three cases to consider. Case 1: $\tau < e.V_{newe}$. In this case, no POS entry is added or removed, since Line 18 through Line 25 have no effect. On the other hand, for the last POS entry with TS=τ, $e.P$ will be removed from its state, accomplished by Line 26 through Line 27. This ensures the content of POS corresponds with P_v, satisfying B1. As POS.Last does not change, B2 continues to hold. By the same argument, B3 and B4 continue to hold as well.

Case 2: $e.V_{newe} \leq \tau < e.V_e$. As is in Case 1, POS.Last.State will have $e.P$ removed by Line 26 through Line 27. However, we show that POS.Last.TS remains unchanged (i.e., POS.Last.TS = τ before and after e is processed) in this case. That is, POS.Last.State.k > 1 before e is processed. This is in turn equivalent to saying that POS.Last.TS is equal to $e'.V_s$ for some e' that was added to POS previously, since in that case POS.Last.State has at least two contributing events, e and e', and therefore k > 1. Suppose for a contradiction that this is not the case. By Invariant τ-S-Algorithms, the only case when τ is set to $e.V_e$ is through an input CTI event, say e''. Let $e''.V_s = t$, and let the τ value after e'' is processed be τ^*. By the semantics of CTI events, $e.V_{newe} \geq t$. We know $\tau^* = e.V_e > e.V_{newe}$. Therefore, $\tau^* > t$. On the other hand, by Algorithm 5, since the value of τ is changed when processing e'', Line 4 in Algorithm 5 is executed. Also, POS.Last.TS before and after processing e'' are different. Therefore, new POS entries are added when processing e''. By Line 1 in Algorithm 5, all the newly added POS entries have TS greater than the τ value before processing e'', but no greater than t. Therefore, after processing e'', POS.Last.TS $\leq t$. However, POS.Last.TS after processing e'' is τ^* by assumption. This is a contradiction.

We have shown POS.Last.TS remains unchanged after processing e. Therefore, B1 through B4 again hold by a similar argument used in Case 1.

Case 3: $\tau \geq e.V_e$. When $\tau > e.V_e$, clearly POS.Last.TS remains unchanged, as the last entry of POS will not be removed when processing e. When $\tau = e.V_e$, it may seem that POS.Last can be removed, if Line 21 is executed. However, by a similar proof-by-contradiction argument as in Case 2, we can show that for the last POS entry, say p, where p.TS = $e.V_e$, p.State.k > 1. Therefore, that entry remains after processing e'. As POS.Last.TS does not changed, B1 through B4 again hold by a similar argument used in Case 1 and Case 2.

Therefore, Invariant POS-τ-S-O holds in all of the three cases.

Invariant τ-O: Given that Invariant POS-τ-S-O holds, by the same argument as in the proof to Lemma 12, τ-O holds as well. □

LEMMA 15. *Algorithm 5 upholds the set of invariants in $\mathcal{I}$.*

PROOF. Invariant ECQ-τ-S: Line 1 removes all tuples in ECQ with TS $\leq e.V_s$. However, this invariant requires that after processing e, ECQ contains exactly all tuples in $\mathcal{S}[\![S]\!]$ with $V_e > \tau_{new}$. Therefore, we need to show that for each tuple r in $\mathcal{S}[\![S]\!]$, $r.V_e > \tau_{new}$ iff $r.V_e > e.V_s$.

There are two cases to consider. Case 1: $e.V_s > \tau_{old}$. In this case, Line 1 may remove tuples from ECQ with TS between $e.V_s$ and τ_{old}. We know in POS before e is processed, POS.Last.TS = τ_{old}. If Line 1 adds new entries to POS, these entries have TS greater than τ_{old}. All POS entries have TS value no greater than $e.V_s$. In this case Line 3 and 4 are executed, so τ_{new} is set to POS.Last.TS. Hence, $\tau_{new} \leq e.V_s$. Therefore, for any tuple r in $\mathcal{S}[\![S]\!]$, if $r.V_e > e.V_s$, $r.V_e > \tau_{new}$. On the other hand, if $r.V_e \leq e.V_s$, r has been added to POS either previously, or while processing e by Line 1. Since POS.Last.TS = τ_{new}, $r.V_e \leq \tau_{new}$. This shows ECQ-τ-S holds.

Case 2: $e.V_s \leq \tau_{old}$. In this case, Line 1 has no effect on ECQ, since by assumption, all tuples in ECQ have TS value greater than τ_{old}. Also, Line 3 and 4 are not executed. So $\tau_{new} = \tau_{old}$. Since this invariant holds on the ECQ content before processing e and τ_{old}, it still holds on the ECQ content after processing e and τ_{new}.

Invariant τ-S-Algorithms: If $e.V_s \leq \tau_{old}$, POS.Last is not changed, and $\tau_{new} = \tau_{old}$, as is argued above. τ' defined in this invariant is no greater than τ_{old}. Therefore, $\tau_{new} = \max(\tau_{old}, \tau')$.

On the other hand, if $e.V_s > \tau_{old}$, $\tau' \geq \tau_{old}$, and $\tau_{new} =$ POS.Last.TS = $\tau' = \max(\tau_{old}, \tau')$ as is argued above. Therefore, this invariant holds.

Invariant POS-τ-S-O: Take any $v \in \mathrm{ENDPT}(\mathcal{S}[\![S_{new}]\!])$. B1 holds for any $v < \tau_{old}$, as the content of these POS entries is not affected by e. For any $\tau_{old} \leq v < \tau_{new}$, new POS entries may be added by Line 1. Each such POS entry p however corresponds to $P_{p.TS}$. So B1 holds. For B2, as is argued above, either the value τ does not change, in which case POS.Last.TS does not change, or τ is explicitly set to POS.Last.TS by Line 4. Therefore, τ_{new} = POS.Last.TS. B3 holds, as those POS entries corresponding to v where $v < e.V_s, v.next < e.V_s$ will have been removed by Line 7. Finally, B4 holds, because when the "while loop" in Line 6 sees such a POS entry p with p.TS $< e.V_s$, p.next.TS $\geq e.V_s$, the loop exists. So this entry becomes POS.First.

Invariant τ-O: Given that Invariant POS-τ-S-O holds, by the same argument as in the proof to Lemma 12, τ-O holds as well. $\square$

Consistency Sensitive Streaming Operators in CEDR

Jonathan Goldstein
Microsoft Research
One Microsoft Way
Redmond, WA 98053
jongold@microsoft.com

Mohamed Ali
Perdue University
mhali@cs.purdue.edu

Roger Barga
Microsoft
One Microsoft Way
Redmond, WA 98053
barga@microsoft.com

Mingsheng Hong
Cornell University
mshong@cs.cornell.com

ABSTRACT

"Consistent Streaming Through Time: A Vision for Event Stream Processing" through a bitemporal model for streaming data systems, introduced a spectrum of consistency levels for handling out of order data. This paper shows how this model may be realized, through the use of speculative output, for select, join, alterlifetime, and sum. In addition, this paper introduces two new operators which can be used, in conjunction with the other operators, to implement the full spectrum of consistency levels. Furthermore, algorithms are given for all these operators which are provably efficient, and close to optimal.

1. INTRODUCTION

Over the past 10 years, many new requirements for streaming and event processing systems have been discovered, and used to design various stream/event processing systems. These requirements derive from a multitude of motivating scenarios, some of which include sensor networks, large scale system administration, internet scale monitoring, and stock ticker data handling. Among the agreed upon requirements are the following:

- Rather than one time queries against static data, queries have a continuous nature, and never terminate (e.g. compute a one minute moving average for heat across a sensor network).
- Insert/event rates are very high (e.g. orders of magnitude higher than a traditional database can process inserts).
- These systems over time have had to handle increasingly expressive standing queries. For instance, today's streaming systems support stateful computation (e.g. join).

In addition, there is a growing consensus in our community that this computation is sufficiently rich that it should be based on compositional SQL/relational algebra semantics, although the precise nature of that basis is in dispute.

While our growing understanding has led to successful streaming systems for specific vertical markets, broad adoption of a single system across a wide spectrum of application domains remains unattained. Rather than a marketing phenomenon, we believe this is a consequence of a missing technical requirement: The domain specific correct handling of out of order data and data retraction. More specifically, consider the following three scenarios:

1. We have a large collection of machines in a corporate network which produces system maintenance events. As a result of transient network phenomena, such as network partitioning, individual events may get arbitrarily delayed. Since the consequence of an alert (e.g. finding machines that didn't come up after a patch was installed) involves human intervention (e.g. a system administrator examining a machine), and is therefore expensive, we should wait for the delayed events to get to the stream processing system before reporting an install problem.
2. We are collecting statistics on web traffic. As in the previous example, networks are unreliable. There is far too much data to remember for any significant period of time, so we simply process the data as it comes in, dropping any significantly late arriving data, and report the best answer we can reasonably compute.
3. We are monitoring a stock ticker for the purpose of computing trades. Occasionally the stock feed provides incorrect data. There is an SLA (service level agreement) in place which gives the data provider 72 hours to report the correct ticker price for each reading. If a stock trade occurs with an incorrect price, the parties have the option to back out of the transaction during that 72 hour period (US SEC requirement). As a consequence, even though results are provided immediately, corrections, which may be posted up to 72 hours later, may lead to some form of compensation. The system must therefore respond instantly, but provide corrections when necessary.

Careful scrutiny of our scenarios reveals 2 interesting aspects of query processing which are being varied:

- How long do we wait before providing an answer (blocking).
- How long do we remember input state both for blocking and for providing necessary compensations once we unblock.

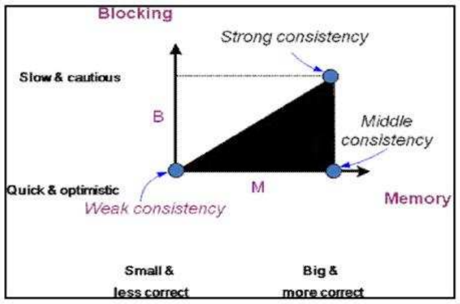

Figure 1 – Spectrum of Possible Consistency Levels

These variables lead to the spectrum of consistency levels described in Figure 1. Our three examples correspond to the three corners of the consistency triangle: High (1), Low (2), Middle (3).

[16] defines these distinct consistency levels formally, based on a bitemporal model for streaming queries that employs both inserts and retractions. In addition, [16] also provided denotational semantics for relational algebra based operators which are explicitly independent of retractions. While no algorithms were introduced in [16], it introduced two notions of correctness for streaming operators which must be enforced using retractions for any valid implementation of the given operator semantics.

In this paper we present the first algorithms for streaming operators which implement the full spectrum of consistency levels by being the first to make use of speculative output. We define correct algorithms for several of these operators, and introduce definitions and algorithms for 3 new operators. In addition, we provide results of a theoretical analysis of all algorithms that show they are provably optimal in many cases and at least close to optimal for others.

Two of the new operators, Align, and Finalize, combined with the other operators, implement the full consistency spectrum shown in Figure 1 over a rich relational algebra based computational model. This is critically important, as it shows that consistency may be varied easily on a *per query basis*.

Section 2 contains a review of the relevant formal model and correctness requirements from [16]. Section 3 introduces the physical event model used in our algorithms, as well as a discussion of processing plans. Section 4 describes our operators and algorithms. Section 5 describes the results of our asymptotic analysis. Section 6 discusses related work. We conclude with Section 7.

2. CEDR Streams and Operator Properties

In this section, we review a variant of our temporal stream model, introduced in [16]. This model is used to characterize CEDR streams, CEDR engine operator semantics, and consistency levels for handling out of order or invalidated data [16]. The temporal model in this paper is simplified from what is described in [16] in the sense that we only model valid time and CEDR time (occurance time is omitted). For the purposes of this paper, this is sufficient since only these two notions of time are necessary to understand CEDR speculative output and consistency levels.

In CEDR, a data stream is modeled as a time varying relation. For most operators, we will use the interpretation that a data stream models a series of updates on the history of a table, in contrast to previous work which models the physical table updates themselves. In CEDR a stream is modeled as an *append only* relation. Each tuple in the relation is an event, and has a logical ID and a payload. Each tuple also has a **validity interval**, which indicates the range of time when the payload is in the underlying table. Similar to the convention in temporal databases, the interval is closed at the beginning, and open at the end. Valid start and end times are denoted as V_s and V_e respectively. When an event arrives at a CEDR stream processing system, its CEDR time, denoted as C, is assigned by the system clock. Since, in general, CEDR systems use different clocks from event providers, valid time and CEDR time are not assumed to be comparable.

CEDR has the ability to introduce the history of new payloads with **insert** events. Since these insert events model the history of the associated payload, they provide both valid start and valid end times. In addition, CEDR streams may also shrink the lifetime of payloads using **retraction** events. These retractions may reduce their associated valid end times, but are not permitted to change their associated valid start times. Retraction events must provide new valid end times, and be uniquely associated with the payloads whose lifetimes are being reduced. A **full retraction** is a retraction where the new valid end time is equal to the valid start time.

The history of a stream can be represented in a *history table* such as the one shown in Table 1. In this table, we have two events. The first event, E0, has its lifetime initially established at CEDR time 1 with payload P1, valid start time of 1, and valid end time of infinity. At CEDR times 2 and 3, the valid end time is retracted first to 10, then to 5. The second event, E1, is initially modeled at CEDR time 3 and has a payload of P2, a valid start time of 4, and a valid end time of 9.

Table 1 – Example of a History Table.

ID	V_s	V_e	C	(Payload)
E0	1	∞	1	P1
E0	1	10	2	P1
E0	1	5	3	P1
E1	4	9	3	P2

A **canonical history table** is a history table in which all retracted rows are removed, all rows whose event IDs are fully retracted are removed, and the C column is projected out. In cases with multiple retractions for the same ID, the order is unambiguous since V_e can only shrink. For instance, in Table 1, both the first and second rows of the table are removed since they are both retracted. Canonical history tables reflect the eventual history of the stream, independent of CEDR arrival time, after retractions have been taken into account.

An **infinite history table** contains all events over all time. Two streams are **logically equivalent** if they have identical infinite canonical history tables.

We now have enough machinery to define our first notion of operator correctness, which applies to all computational operators (as opposed to operators whose purpose is to vary consistency):

Definition 1: A CEDR operator O is **well behaved** iff for all (combinations of) inputs to O which are logically equivalent, O's outputs are also logically equivalent.

Intuitively, the above definition says that a CEDR operator is well behaved as long as the output produced by the operator semantically converges to the output produced by a perfect version of the input without retractions and out of order delivery.

Our second notion of operator correctness only applies to operators which are based on materialized view update semantics. While a formal definition is provided in [16], we provide a more intuitive definition here:

Definition 2: Assuming we interpret the stream as modeling the changes to a relation, and the valid time intervals as describing the time (V_s) at which the payload was inserted, and the time (V_e) at which the payload was removed, a **view update compliant** operator produces snapshot identical output for snapshot identical input. In other words if the table contents are identical for all snapshots of two inputs, the output snapshots must also match.

[16] specified the denotational semantics of a number of operators based on relational algebra. For brevity, we will only provide algorithms for three view update compliant operators. A stateless operator (select), a join based operator (equijoin), and an aggregation based operator (sum). In these definitions, E(S) is the set of events in the infinite canonical history table for stream S.

Selection corresponds exactly to relational selection. It takes a Boolean function f which operates over the payload. The definition follows:

Definition 3: Selection $\sigma_f(S)$:

$\sigma_f(S)=\{(e.V_s, e.V_e, e.Payload) \mid e \in E(S) \text{ where } f(e.Payload)\}$

Similarly, the next operator is **join**, which takes a boolean function θ over two input payloads:

Definition 4: Join $\bowtie_{f(P1,P2)}(S_1, S_2)$:

$\bowtie_{\theta(P1,P2)}(S_1, S_2) = \{(V_s, V_e, (e_1.Payload \text{ concantenated with } e_2.Payload)) \mid e_1 \in E(S_1), e_2 \in E(S_2), V_s=\max\{e_1.V_s, e_2.V_s\}, V_e=\min\{e_1.V_e, e_2.V_e\}, \text{ where } V_s < V_e, \text{ and } \theta(e_1.Payload, e_2.Payload)\}$

Intuitively, the definition of join semantically treats the input streams as changing relations, where the valid time intervals are the intervals during which the payloads are in their respective relations. The output of the join describes the changing state of a view which joins the two input relations. In this sense, many of our operators follow view update semantics such as those specified in [14].

The last materialized view compliant operator is introduced in this paper, and is the first aggregate for which we provide denotational semantics. **Sum** sums the values of a given column for all rows in each snapshot, starting at the earliest possible time. An observant reader will note that the given definition is implementable without retractions if there are no retractions in the input, and all events arrive in V_s order. More specifically, we only output sums associated with snapshots which precede the arriving event's V_s. Note that the output event lifetimes have valid start and end points which are determined by the valid start and end points of the input events. This is sensible given that the output sum values may only change when an input tuple is added or removed from the modeled input relation. The definition for sum follows:

Definition 5: $Sum_A(S)$:

$C = \{e.V_s \mid e \in S\} \cup \{e.V_e \mid e \in S\} \cup \{0\}$

Let C[i] be the ith earliest element of C

$sum_A(S) = \{(V_s, V_e; a) \mid |C| > t >= 1, V_s=C[t], V_e=C[t+1], a= \Sigma_{e \in S, e.Vs <= Vs, Ve <= e.Ve} e.A\}$

While all CEDR computational operators are well behaved, not all are view update compliant. Indeed, the streaming only operators which our community has discovered (e.g. windows, deletion removal) are not view update compliant by necessity. In CEDR, we can easily model these operators with **AlterLifetime**. AlterLifetime takes two input functions, $f_{Vs}(e)$ and $f_\Delta(e)$. Intuitively, Alterlifetime maps the events from one valid time domain to another. In the new domain, the new V_s times are computed from f_{Vs}, and the durations of the event lifetimes are computed from f_Δ. The precise definition follows:

Definition 6: AlterLifetime $\Pi_{fvs, f\Delta}(S)$

$\Pi_{fvs, f\Delta}(S)=\{(|f_{Vs}(e)|, |f_{Vs}(e)| + |f_\Delta(e)|, e.Payload) \mid e \in E(S)\}$

From a view update compliant operator's perspective, AlterLifetime has the effect of reassigning the snapshots to which various payloads belong. We can therefore use AlterLifetime to reduce a query which crosses snapshot boundaries, like computing a moving average of a sensor value, to a problem which computes results within individual snapshots, and is therefore view update compliant. For instance, [16] noted that a moving window operator, denoted W, is a special instance of Π. This operator takes a window length parameter wl, and assigns the validity interval of its input based on wl. More precisely: $W_{wl}(S) = \Pi_{Vs,wl}(S)$. Once we use AlterLifetime in this manner, each snapshot of the result contains all tuples which contribute to the windowed computation at that snapshot's point in time. Therefore, when we feed this output to sum, the result is a moving sum with window length wl.

One can similarly define hopping windows using integer division. Finally, we can even use the AlterLifetime operator to easily get all inserts and deletes from a stream:

- $Inserts(S)= \Pi_{Vs, \infty}(S)$
- $Deletes(S)= \Pi_{Ve, \infty}(S)$

In addition this paper introduces two new operators, called **align** and **finalize** which while uninteresting from a computational model point of view, are used to implement the full spectrum of consistency levels. We define them fully in Section 4.

3. CEDR Physical Stream Model

While the stream model in Section 2 is a useful theoretical construct for formally defining the semantics of streams and operators, the operator algorithms themselves are based on a slightly different definition of a stream. More specifically, operators respond to individual events as they arrive at the CEDR system. While CEDR time is implicitly encoded in the event arrival order, it is not explicitly part of a CEDR physical event.

CEDR operators receive, one at a time, three types of events. The first type of event is an insert, which corresponds semantically to insert events in the CEDR bitemporal model. Insert events come with V_s and V_e timestamps, and also a payload. Note that CEDR uses bag semantics, and may, therefore, receive two inserts with identical payloads and identical lifespans.

The second type of event is a retraction, which corresponds semantically to retractions in the CEDR bitemporal model. Since retractions must be paired with their corresponding inserts or previous retractions, we either need to have global event IDs, or include in the retraction enough information to establish the pairing. If we used global IDs, certain stateless operators, like select, would become more complicated. Since we consider retractions to be far less common than inserts, we will instead include all necessary information in the retraction to establish the connection with the original insert. Note, however, that the algorithms presented in this paper may be easily adapted to make use of global IDs if desirable. CEDR physical retractions therefore include the original valid time interval, V_s, and V_e, the new end valid time V_{newe}, and the payload values from the original insert.

Note that the physical stream associated with the logical stream in Table 1 is given in Table 2 below:

Table 2: Physical Stream Representation

Event Type	V_s	V_e	V_{Newe}	(Payload)
Insert	1	∞		P1
Retract	1	∞	10	P1
Retract	1	10	5	P1
Insert	4	9		P2

The third type of event is a kind of punctuation. This type of event, called a **CTI** (current time increment), comes with a timestamp V_e. The semantics of the message are that all events have arrived in the stream whose sync times ([16]) are less than the accompanying timestamp. More specifically, the sync times for insert events occur at V_s, while the sync times for retraction events occur at V_{newe}.

There are actually two types of CTIs. The first type is an internal CTI, which we assume cannot be reordered to a position in the stream prior to its earliest correct placement. This corresponds to the CTI described in the earlier paragraph. The second type of CTI, called an ExternalCTI, may arrive arbitrarily out of order relative to the rest of the stream contents. We only define the handling of ExternalCTIs for Finalize, which converts out of order externalCTIs into in order internal CTIs. External CTIs have a V_s, a V_e, and a Count. The semantics are that Count events exist in the stream whose sync times are in the timestamp interval $[V_s,V_e)$. Furthermore, while ExternalCTIs may arrive arbitrarily out of order, they must have nonoverlapping valid time intervals.

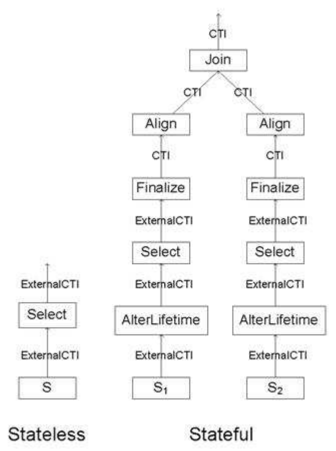

Figure 2: CEDR Processing Plans

Two typical CEDR query plans are shown in Figure 2. Note that in these plans, Finalize has two purposes. First, it forces the purging of state in a stateful operator by issuing CTIs and therefore controls the forgetfulness axis of consistency. Second, it partially reorders external streams, which may be arbitrarily out of order, into well behaved internal CEDR streams. More specifically, Finalize ensures that all output CTIs are output no earlier than the earliest correct time, and also ensures that all retractions in a retraction chain are issued in correct relative order.

The second operator to note is the Align operator. This operator blocks the events of the incoming stream and combines inserts and retractions with other retractions when possible. The blocking time is controlled by a provided input function, although internal CTIs may cause early unblocking. The Align operator, therefore, controls the blocking axis of consistency.

Note that there are two types of plans in Figure 2. The first type of plan is a stateless plan, and doesn't have either a Finalize or Align operator. These operators are unnecessary in this plan since there is no state to purge, and nothing is gained by blocking out of order events. In this plan, since we never convert ExternalCTIs to CTIs, we simply allow the ExternalCTIs to pass through the operators unchanged. This is generally the case with stateful operators, with the one exception, AlterLifetime, which we discuss later. We will not discuss ExternalCTI algorithms for other stateless operators.

The more interesting type of plan is the stateful plan, which contains, below the first stateful operator, a Finalize, followed by an Align. These two operators together determine the consistency of the query. Note, all stateful plans must minimally have a Finalize somewhere below the first stateful operator, although an align is optional. This is to ensure that all stateful operators operate over streams with no out of order CTIs and no out of order delete chains. As a result, no ExternalCTI algorithms are specified for stateful operators, with the exception of Finalize.

4. CEDR Operator Algorithms

Operators in the CEDR system have a 1 to 1 correspondence with the operators discussed in Section 2. These operators must faithfully implement the denotational semantics provided in all cases in order to be considered correct. While formal correctness proofs are beyond the scope of this work, we will provide informal arguments as to the correctness of the provided algorithms, as well as detailed examples for Join and Finalize showing the algorithm behavior on various input streams.

All operators, except align and finalize, are written assuming an output consistency level which involves no blocking and infinite memory ($B=0$, $M=\infty$). Algorithms for CTI events will, however, clean state which is sufficiently stale that the CTI and operator semantics guarantee the state is no longer needed. Operators implemented in this manner can be made to behave according to any consistency level using align and finalize.

All operators are written using copy out semantics. While this has no effect on asymptotic behavior, a real system would need to avoid this where possible. All provided algorithms may be adapted to avoid copying in many places but such a discussion is beyond the scope of this paper.

For each operator, we provide the behavior for processing insert, retraction, and CTI events, and for some operators, ExternalCTI events. Whether this processing is the result of a push or pull model of event processing is not relevant to our discussion, which is about the processing algorithms themselves. Clearly a full system implementation needs to address architectural issues such as push versus pull event processing and operator scheduling.

Throughout the algorithms, references are made to data structures which are ordered according to a specified key. These structures have a number of methods:

Collection.Insert(K) inserts a key, in some cases a key value pair

- Collection.Remove(K) removes a key and possibly an accompanying value from the structure
- ResultCursor = Collection.Search(K) returns a cursor into the structure which initially points to the first exact key match. If there is no match, ResultCursor = Empty
- ResultCursor = Collection.SearchL(K) similar to Search but returns a pointer to the first key less than K. Similarly we have SearchLE, SearchGE, and SearchG.

- ResultCursor = Collection.First() returns the first element of the collection according to the given sort order

In addition, some operators make use of interval search structures based on a multidimensional structure. These structures have Insert, Remove, and Search functions comparable to the ones previously described for unidimensional structures. When Search is given a point, it returns all intervals which contain the point. When it is given an interval, it finds all overlapping intervals.

Inequality based searches can only be performed with an input point, and are only used in situations where data intervals are nonoverlapping, and therefore ordered.

4.1 Select

Select is the most straightforward operator described in this paper. It is a simple filter which allows inserts and retractions to pass through the operator unchanged if they satisfy a Boolean function f(Payload). Note that retractions need no special handling since the payload is included in the retraction. Therefore, if the payload in the retraction satisfies the function, the matching retraction chain has already passed through the select. The resulting algorithms follow:

```
Algorithm for $\sigma_f$(S):

Insert/Retraction(e):
If f(e.Payload)
  Output a copy of e

CTI(e):
Output a copy of e
```

4.2 AlterLifetime

This operator uses two provided functions, $f_{Vs}(e)$ and $f_\Delta(e)$, to map incoming events from one valid time domain to another. In order to make AlterLifetime implementable, we have a few important constraints on these input functions:

- $|f_{Vs}(e)|$ must be constant or increasing with increasing V_s, and may only depend on V_s and constants (e.g. window size, chronon, etc…)
- $|f_\Delta(e)|$ must be constant or decreasing with decreasing V_e

The first constraint ensures that CTIs in the input imply CTIs on the output. The last constraint ensures that retractions in the input will never produce event lifetime expansions in the output.

It may seem that AlterLifetime is like select, and merely needs to pass events through with their lifetimes modified according to the input functions, but there is an important special case to consider. Since full retractions and their associated events are removed from the infinite canonical history tables upon which the semantics of AlterLifetime are defined, we must ensure that full retractions in the input lead to full retractions in the output in all cases. This should be true even if, for instance, $f_\Delta(e)$ is a constant, which is a common function and implements windows. The resulting algorithms for insert and retraction are shown below. Note that in this algorithm $f_{new\Delta}(e)$ refers to a version of $f_\Delta(e)$ where all references to V_e are replaced with references to V_{newe}.

Algorithm for $\Pi_{fvs, f\Delta}(S)$:

Operator state:

- LastCTI is a timestamp variable initialized to 0

```
Insert(e):
Create an insert event ie
ie.$V_s$ = $|f_{Vs}(e)|$
ie.$V_e$ = $|f_{Vs}(e)| + |f_\Delta(e)|$
ie.Payload = e.Payload
output ie

Retraction(e):
Create a retraction event re
re.$V_s$ = $|f_{Vs}(e)|$
re.$V_e$ = $|f_{Vs}(e)| + |f_\Delta(e)|$
re.Payload = e.Payload
If e is a full retraction
  re.$V_{newe}$ = $|f_{Vs}(e)|$
Else
  re.$V_{newe}$ = $|f_{Vs}(e)| + |f_{new\Delta}(e)|$
Output re
```

Now we discuss the algorithm for CTI. In this discussion, $f_{Ve}(e)$ refers to a version of $f_{Vs}(e)$ where all references to V_s are replaced with references to V_e. The algorithm for CTI is also not as simple as it might first seem. If we always generate an output event, using $V_e=|f_{Ve}(e)|$, we might generate CTIs which do not advance the clock. Since we assume that CTIs arrive in increasing V_s order, we can avoid this problem by delaying the output CTI until we receive a CTI which moves the output CTI forward in time. The resulting algorithm follows:

```
CTI(e):
If LastCTI != $|f_{Ve}(e)|$
  Create a CTI event ctie
  ctie.$V_e$=$|f_{Ve}(e)|$
  LastCTI = ctie.$V_e$
  output ctie
```

While there is no algorithm for external CTI, the given algorithms may be integrated into a combination of Finalize and AlterLifetime when it is desirable to combine windowing with forced expiration to improve state management.

4.3 Equijoin

Equijoins are joins (See Section 2) where the Boolean function θ, when put in conjunctive normal form (CNF), has one or more conjuncts which are equality tests on columns from both input streams. For instance, consider the join plan:

$$\bowtie_{S1.P = S2.P}(S_1, S_2)$$

Actual input streams are shown in Table 1, which we use to illustrate how join works. Note that the table includes both input streams, and uses an Sid column to distinguish between events from the different streams. In addition, we include a CEDR time column, even though it's not part of the physical event, to establish the order of arrival. Also note that Table 2 shows the infinite canonical history table for the input streams, and Table 3 shows the result of applying the denotational semantics of the join to the infinite canonical history tables. For our join algorithm to be correct, the infinite canonical history table of the output must be identical to Table 3.

Table 1 : Physical input streams for join

Sid	Type	V_s	V_e	V_{newe}	C	P
S_1	Insert	0	2		1	A_0

S_1	Cti		1		2	
S_1	Insert	2	6		3	A_1
S_2	Insert	3	5		4	A_1
S_2	Cti		3		5	
S_1	Retract	2	6	4	6	A_1

Table 2 : Infinite Canonical History Table of Input

sid	V_s	V_e	(Payload)
S_1	0	1	A_0
S_1	2	4	A_1
S_2	3	5	A_1

Table 3 : Infinite Canonical History Table of Output

V_s	V_e	(Payload)
3	4	A1

Our algorithm for join is based on symmetric hash join. When an event arrives on one side, the other side is checked and output is produced. If the incoming event is an insert, we join to the other side and output necessary inserts. If the incoming event is a retraction, we join to the other side to see whether any retractions of previously output events are necessary. Note that in addition to checking the equality predicate, for inserts, we also need to check whether join candidates have lifetimes which overlap the lifetime of the incoming event. In the case of retractions, we need to check whether the result of applying the retraction causes a previously output lifetime to shorten. Both of these checks involve retrieving all the entries from the hash bucket whose lifetimes overlap the input event. We will therefore use a multidimensional structure to perform the overlaps test in an algorithmically efficient manner.

The resulting algorithms for insert and retraction are shown below. Note that since our algorithm is symmetric, we consider events only on S_1.

Algorithm for $\bowtie_{\theta(P1,P2)}(S_1, S_2)$:

Operator state:

- 2 Hashtables $SHash_1$ and $SHash_2$ which hash on the columnsets of the equijoin on S_2 and S_1 respectively. Each hash bucket contains a multidimensional structure keyed on the valid time interval of the event. Each hash supports three methods. Insert(e) and Remove(e) respectively add and remove events to the two level structure. FindMatchingInsert(e) takes a retraction event and returns the insert event in the two level structure which pairs with the retraction. $SHash_1$ and $SHash_2$ are initially empty.
- S_1CTI and S_2CTI are timestamp variables which hold the latest CTI V_e from S_1 and S_2 respectively and are initialized to 0.
- Ordered structures $EventV_eQ_1$ and $EventV_eQ_2$ with <key, value> = <V_e, event>. These are used for cleaning state when we receive CTI events.

```
Insert on S1(e):
ResultCursor = SHash2.Lookup(e).Search(e.Vs,
e.Ve)
While ResultCursor != Empty
  If θ(e.Payload,ResultCursor.event.Payload)
    Create an insert event ie
    ie.Vs = max (e.Vs, ResultCursor.event.Vs)
    ie.Ve = min (e.Ve, ResultCursor.event.Ve)
    ie.Payload = (e.Payload,
                  ResultCursor.event.Payload)
    output ie
  Increment ResultCursor
If e.Ve >= S2CTI
  SHash1.Insert(e)
  EventVeQ1.Insert(e.Ve, e)
```

```
Retraction on S1(e):
ResultCursor = SHash2.Lookup(e).Search(e.Vs,
e.Ve)
While ResultCursor != Empty
  If θ(e.Payload,ResultCursor.event.Payload)
     and
     e.Vnewe<min(e.Ve, ResultCursor.event.Ve)
    Create a retraction event re
    re.Vs = max (e.Vs, ResultCursor.event.Vs)
    re.Ve = min (e.Ve, ResultCursor.event.Ve)
    re.Vnewe =
           max(e.Vnewe, ResultCursor.event.Vs)
    re.Payload = (e.Payload,
                  ResultCursor.event.Payload)
    output re
  Increment ResultCursor
ie = SHash1.FindMatchingInsert(e)
if ie != NULL
  SHash1.Remove(ie)
  EventVeQ1.Remove(ie.Ve, ie)
  If e.Vnewe >= S2CTI
    ie.Ve = e.Vnewe
    SHash1.Insert(ie)
    EventVeQ1.Insert(ie.Ve, ie)
```

The algorithm for CTI events is quite simple. When the min of S_1CTI and S_2CTI increases, we output a CTI. In addition we remove events in our state that can no longer contribute to future results. The algorithm is shown in Figure … Since this algorithm is also symmetric, we will again consider events only on S_1.

```
CTI on S1(e):
If S1CTI < S2CTI
  Create a CTI event ctie
  ctie.Ve= min(e.Ve, S2CTI)
  output ctie
S1CTI = e.Ve
While EventVeQ2.NotEmpty &&
      EventVeQ2.First().Ve < e.Ve
  eventtoageout = EventVeQ2.First()
  EventVeQ2.Remove(eventtoageout)
  SHash2.Remove(eventtoageout)
```

Observe that this join algorithm, when provided with the input in Table 1, produces the output in Table 4. The infinite canonical history table of this output is, as required by the denotational semantics of join, the same as Table 3.

Table 4: Physical output of Join

Type	V_s	V_e	V_{newe}	P
Insert	3	5		A_1
CTI		1		
Retract	3	5	4	A_1

4.4 Sum

See accompanying theory paper.

4.5 Align

While the align operator is a pass through from a denotational semantics point of view (the input and output infinite canonical history tables are identical), it is a vital component for realizing the spectrum of consistency levels described in the introduction. Specifically, Align is used to adjust the blocking component of consistency.

This is accomplished by buffering and blocking incoming events for a certain period of time. We will not specify in our algorithm whether we are blocking based on application time or system time, as either option is easily implementable and the distinction is semantically unimportant. While the events are buffered, any retractions are combined with buffered earlier inserts or retractions of the same event. When events are unblocked through a CTI, events are released in sync timestamp order, and are accompanied by an output CTI.

Another way the operator may become unblocked is by using the outputtime() function, which returns the latest application timestamp of events which should be unblocked. This function may internally refer to either system or application time. The only requirement is that outputtime() must stay constant or increase with subsequent calls. When streams are unblocked in this manner, the V_s time, rather than the sync time, is used, and no CTI is issued. This is due to the assumption that once an event is unblocked, all subsequent retractions for that event should also be unblocked.

Note that any stream may be converted to the highest consistency level from a blocking point of view by having outputtime() always return 0. All Align algorithms make use of a method, called Unblock. Unblock is the routine which actually releases blocked events in accordance with outputtime(). The algorithms for insert and retraction are provided in below:

Algorithm for $\text{Align}_{\text{outputtime()}}(S)$:

Operator state:

- CurrentOutputTime keeps the last reading of outputtime(), and represents the latest V_s for which output events have been unblocked. CurrentOutputTime is initially set to 0.
- LastCTI is the timestamp value of the last output CTI issued. It is initialized to 0
- bufferedinserts is an ordered data structure which buffers events and uses the ordering key (V_s, V_e, P). This structure may be searched using a retraction event r. This search returns a match if the buffered entry b matches in the following way: $b.V_s = r.V_s$, $b.V_e = r.V_e$, b.Payload = r.Payload
- bufferedretractions is an ordered data structure which buffers events and uses the ordering key (V_s, V_{newe}, P, V_e). This structure may be searched using a retraction event r. This search returns a match if the buffered entry b matches in the following way: $b.V_s = r.V_s$, $b.V_{newe} = r.V_e$, b.Payload = r.Payload
- Ordered structure $\text{EventV}_e\text{Q}_{\text{inserts}}$ which contains insert events ordered by (V_e, V_s, P). This is used for unblocking inserts when we receive CTI events.
- Priority queue $\text{EventV}_{\text{newe}}\text{Q}_{\text{retractions}}$ which contains retraction events ordered by (V_{newe}, V_e, V_s, P). This is used for unblocking retractions when we receive CTI events.

```
Unblock():
CurrentOutputTime = outputtime()
While ((bufferedinserts.First() != NULL) &&
       (bufferedinserts.First().Vs <=
        CurrentOutputTime)) or
      ((bufferedretractions.First() !=
       NULL) &&
       (bufferedretractions.First().Vs <=
        CurrentOutputTime))
  if ((bufferedretractions.First() ==
      NULL) ||
     ((bufferedinserts.First() != NULL) &&
      (bufferedinserts.First().Vs <=
       bufferedretractions.First().Vs)
    eb = bufferedinserts.First();
    Output a copy of eb
    bufferedinserts.Remove(eb)
    EventVeQinserts.Remove(eb)
  else
    eb = bufferedretrations.First();
    Output a copy of eb
    bufferedretractions.Remove(eb)
    EventVneweQretractions.Remove(eb)

Insert(e):
bufferedinserts.Insert(e)
EventVeQinserts.Insert(e)
Unblock()

Retraction(e):
ResultCursor = bufferedinserts.Search(e)
If ResultCursor != Empty
  etemp = ResultCursor.event()
  bufferedinserts.Remove(etemp)
  EventVeQinserts.Remove(etemp)
  etemp.Ve = e.Vnewe
  bufferedinserts.Insert(etemp)
  EventVeQinserts.Insert(etemp)
else
  ResultCursor=bufferedretractions.Search(e)
  If ResultCursor != Empty
    etemp = ResultCursor.event()
    bufferedretractions.Remove(etemp)
    EventVneweQretractions.Remove(etemp)
    etemp.Vnewe = e.Vnewe
    bufferedretractions.Insert(etemp)
    EventVneweQretractions.Insert(etemp)
  else
    bufferedretractions.Insert(e)
    EventVneweQretractions.Insert(e)
Unblock()
```

Align may also be unblocked when it receives a CTI. This is permissible, even if outputtimer() hasn't reached the incoming CTI's V_e since we have a guarantee that no more events can arrive which may be combined with some buffered events. The algorithm can be found below:

```
CTI(e):
While ((EventV$_e$Q$_{inserts}$.First() != NULL) &&
        (EventV$_e$Q$_{inserts}$.First().V$_e$ <= e.V$_e$)) or
       ((EventV$_{newe}$Q$_{retractions}$.First() != NULL)&&
        (EventV$_{newe}$Q$_{retractions}$.First().V$_{newe}$ <=
         e.V$_e$))
  if ((EventV$_{newe}$Q$_{retractions}$.First() == NULL) ||
       ((EventV$_e$Q$_{inserts}$.First() != NULL) &&
        (EventV$_e$Q$_{inserts}$.First().V$_e$ <=
         EventV$_{newe}$Q$_{retractions}$.First().V$_{newe}$)
    e$_b$ = EventV$_e$Q$_{inserts}$.First()
    Output copy of e$_b$
    bufferedinserts.Remove(e$_b$)
    EventV$_e$Q$_{inserts}$.Remove(e$_b$)
  else
    e$_b$ = EventV$_{newe}$Q$_{retractions}$.First()
    Output copy of e$_b$
    bufferedretractions.Remove(e$_b$)
    EventV$_{newe}$Q$_{retractions}$.Remove(e$_b$)
Unblock()
If bufferedinserts.First() != NULL
  NewCTI=min(e.V$_s$,bufferedinserts.First().V$_s$)
Else
  NewCTI=e.V$_s$
If (NewCTI > LastCTI)
  Output a CTI with V$_s$ = NewCTI
  LastCTI = NewCTI
```

4.6 Finalize

The Finalize operator serves two related purposes in the system. The first purpose is to enable queries with Finalize operators to navigate in the consistency space along the memory axis. This is accomplished by having Finalize issue CTIs for time periods which we are willing to call "final". These CTIs are induced by a function finalizetime(), which is similar to the outputtime() function used in Align. Unlike Align, Finalize does impact the infinite canonical history table of the output, as any operator that limits memory must. It should therefore be used with a high degree of care and understanding.

The second purpose of finalize is, like align, semantically transparent. Finalize buffers all incoming events, with the purpose of correctly ordering out of order retraction chains, and also placing output CTIs at no earlier than the earliest correct opportunity. It receives ExternalCTIs instead of CTIs. While it never blocks well formed output, it does have to remember most events until either an external CTI guarantees it can no longer connect to an incoming retraction, or our finalizetime() function enables us to forget them. These events are stored in a structure called receivedevents, which is sorted on the events' sync time. These algorithms are broken into a number of pieces. In addition to the usual insert, retraction, and CTI functions, we also have a function CleanState which removes all events from supporting memory structures with sync time less than the time passed into the function. We also have a function AgeOut which forces CTIs and event cleanup based on the time returned by finalizetime(). Finally, we have ReceiveEventforCTI which adds events to receivedevents and also checks to see whether we can correctly issue a CTI. The algorithms for insert, retraction, and CTI can be found below:

Algorithm for Finalize$_{finalizetime()}$(S):

Operator state:

- LastCTI is the timestamp value of the last output CTI issued. It is initialized to 0
- CTILifetimes is an ordered data structure which stores non-overlapping valid time intervals [V_s, V_e) in order. Associated with each entry are expected and received fields, which store the number of events expected and received with a sync time which falls within the associated time interval. When insert is called, the parameters provide initial values in the following order (V_s, V_e, expected, received)
- Receivedevents is an ordered data structure with entries <key, value> = <V_{sync}, Event>
- brokenretractionssucc is an ordered data structure which stores entries <Key, value> = <(V_s, V_e, Payload), Event>. When it is searched using a retraction r, it finds all entries e_b with $e_b.V_s = r.V_s$, $e_b.V_e = r.V_{newe}$, e_b.Payload = r.Payload
- Brokenretractionsprev is an ordered data structure which stores entries <Key, Value> = <(V_{newe}, V_s, Payload), Event>. When it is searched using a retraction r, it finds all entries e_b with $e_b.V_s = r.V_s$, $e_b.V_{newe} = r.V_e$, e_b.Payload = r.Payload
- bufferedinsertsbyV_s is an ordered data structure which stores entries <Key> = < V_e, V_s, Payload>. The entries correspond to a set of inserts and may be searched with either an insert or retraction. In either case, exact field matches will be searched for, and in the case of retractions, V_{newe} is ignored.

```
UpdateCTI(NewCTI)
If LastCTI < NewCTI
  Create CTI event ctie
  ctie.V$_e$ = NewCTI
  CleanState(ctie.V$_e$)
  output ctie
  LastCTI = ctie.V$_e$
```

```
ReceiveEventforCTI(V$_{sync}$, e)
receivedevents.Insert(V$_{sync}$, e)
ResultCursor = CTILifetimes.Search(V$_{sync}$)
If ResultCursor != Empty
  Increment ResultCursor.received
  If ResultCursor.received=
     ResultCursor.expected and
     ResultCursor.Entry() =
     CTILifetimes.First()
    UpdateCTI(ResultCursor.V$_e$)
```

```
CleanState(V$_{stale}$)
For each event e in receivedevents with V$_{sync}$
< V$_{stale}$:
  remove entry from receivedevents


For each
```

```
  If e is a retraction
    brokenretractionssucc.Remove(e)
    brokenretractionsprev.Remove(e)
  Else
    bufferedinsertsby$V_s$.Remove(e)
```

```
While CTILifetimes.First().$V_e$ <= $V_{stale}$
  CTILifetimes.Remove(CTILifetimes.First())
```

```
AgeOut($V_{stale}$)
ResultCursor = CTILifetimes.Search($V_{stale}$)
If ResultCursor != Empty
  NewCTI = ResultCursor.$V_s$
Else
  NewCTI = $V_{stale}$
UpdateCTI(NewCTI)
```

```
Insert(e):
If e.$V_s$ >= LastCTI
  $e_{temp}$.$V_s$ = e.$V_s$
  $e_{temp}$.$V_{newe}$ = e.$V_e$
  $e_{temp}$.Payload = e.Payload
  ResultCursor =
          brokenretractionssucc.Search($e_{temp}$)
  if ResultCursor != Empty
    e.$V_e$ = ResultCursor.$V_{newe}$
    $e_{remove}$ = ResultCursor.Event()
    brokenretractionssucc.Remove($e_{remove}$)
    brokenretractionsprev.Remove($e_{remove}$)
  Output a copy of e
  bufferedinsertsby$V_s$.Insert(e)
  ReceiveEventforCTI(e.$V_s$, e)
  AgeOut(finalizetime())
```

```
Retraction(e):
If e.$V_{newe}$ >= LastCTI
  ResultCursor=
             brokenretractionsprev.Search(e)
  if ResultCursor != Empty
    e.$V_e$ = ResultCursor.$V_e$
    $e_{remove}$ = ResultCursor.Event()
    brokenretractionssucc.Remove($e_{remove}$)
    brokenretractionsprev.Remove($e_{remove}$)
  ResultCursor =
             Brokenretractionssucc.Search(e)
  if ResultCursor != Empty
    e.$V_{newe}$ = ResultCursor.$V_{newe}$
    $e_{remove}$ = ResultCursor.Event()
    brokenretractionssucc.Remove($e_{remove}$)
    brokenretractionsprev.Remove($e_{remove}$)
  bufferedinsertsby$V_s$.Search(e)
  if ResultCursor = Empty
    brokenretractionsprev.Insert(e)
    brokenretractionssucc.Insert(e)
  else
    ResultCursor.$V_e$ = e.$V_{newe}$
    output a copy of e
  ReceiveEventforCTI(e.$V_{newe}$, e)
  AgeOut(finalizetime())
```

```
ExternalCTI(e)
CTILifetimes.Insert(e.$V_s$, e.$V_e$, e.Count, 0)
ResultCursor = CTILifetimes.Search(e.$V_s$)
For each element $e_r$ of receivedevents with
  e.$V_s$<= $V_{sync}$ < e.$V_e$
  Increment ResultCursor.received
  If ResultCursor.received =
     ResultCursor.expected and
     ResultCursor.entry() =
     CTILifetimes.First()
         UpdateCTI(ResultCursor.$V_e$)
  AgeOut(finalizetime())
```

To better understand the subtle behavior of this important operator, we include an example to illustrate how broken retraction chains are repaired, and how internal CTIs are generated from external CTIs. The physical stream corresponding to this example is shown in Table 5. Note that a CEDR time column is included to show the order of arrival, though it is not part of the physical event. In addition, we show the Count field for ExternalCTIs in the payload column. When we refer to event 1, we refer to the event which arrived at CEDR time 1. When we refer to event 2, we refer to the event which arrived at CEDR time 2 etc…

Table 5 : Physical Input Stream for Finalize

Type	V_s	V_e	V_{newe}	C	P
Retract	0	10	8	1	P_0
Retract	0	6	4	2	P_0
Retract	0	8	6	3	P_0
Insert	0	10		4	P_0
ExternalCTI	0	8		5	5
Insert	1	5		6	P_1

We assume the finalizetime() function associated with this finalize operator always returns 0. In this case, finalize will not change the semantics (eventual state) of its input stream. Now we will show that the algorithm for finalize produces the correct result.

- When event 1 arrives, it is inserted into brokenretractions as well as receivedevents.
- When event 2 arrives, it is also inserted into brokenretractions and receivedevents.
- When event 3 arrives, it is combined with the broken retractions from events 1 and 2 and itself stored in brokenretractions with V_s=0, V_{newe}=4, V_e=10. Events 1 and 2 are removed from brokenretractions. The original event 3 is also inserted into receivedevents.
- When event 4 arrives, it is combined with the event in brokenretractions, and its.V_e is changed to 4. The event in brokenretractions is then removed. Also, the modified event 4 is output and inserted into bufferedinsertsbyV_s as well as receivedevents.
- When externalCTI arrives at CEDR time 5, an entry, denoted as N, with value (0, 8, 5, 0) is inserted into CTILifetimes. Next, since there are 4 events stored in receivedevents with sync value between 0 and 8, N.received is set to 4.
- When event 6 arrives, it is first output, and then stored in both bufferedinsertsbyV_s and receivedevents. Next, since event 6 falls into entry N in CTILifetimes. N.received is

incremented, and is now equal to N.expected=5. As a result, an internal CTI event is output with V_e = 8. LastCTI is updated to 8, and in cleaning the operator state, we remove all 5 events stored in receivedevents, the two events stored in bufferedinsertsbyV_s, and N in CTILifetimes.

5. Asymptotic Results

This section presents our asymptotic algorithm complexity results. All analysis is worst case in terms of the number of input tuples n, and the number of output tuples m. The complexity results are presented in this manner in order to better understand the optimality of the algorithms presented in this paper. For instance, even though the worst case time complexity for join is inherently quadratic in the size of the input, this bad case only occurs when the output is similarly large. O(n+m) then becomes a lower bound for the time complexity of all algorithms since all input must be read and all output must be produced.

The results for the worst cases are the following:

- For stateless operators, all operators have time complexity of O(n+m), which is clearly optimal. Space complexity is also optimal at O(1).
- For stateful operators, all operators have worst case time complexity bounded by O((logn)(n+m)). Worst case space complexity is O(nlogn), which while clearly not optimal (O(n) is optimal), enables sub-quadratic complexity for time.

An interesting special case to consider is the case where state is bounded, as is typically the case with well behaved streams when operators are windowed. In this case, all operators have time complexity of O(n+m) and space complexity of O(1), which is clearly optimal. The log in the general case comes from having to fix output arbitrarily far back in time, which requires an unbounded accumulation of state, over time. The log is associated with traversing an ordered structure (in some cases multidimensional), through internal nodes of this stored state.

6. Related Work

Motivated by enterprise activities [1], RFID technologies [2], sensor networks [3], surveillance systems [4], network traffic management [5][20], and many other applications, the research focus of the database community has been directed to event stream processing. Several research efforts have been conducted to develop an expressive language for event stream processing [6][7][8][16][20] and efficiently process continuous queries over the incoming streams of event data [9][10][11]. Along this research direction, several data stream management systems have been developed, e.g., [12-19][20]. In this section, we overview related work and differentiate the CEDR system [16] from existing approaches with respect two major issues: (1) The stream representation model and (2) the corresponding consistency guarantees provided by the system's query processor.

Stream tuples are modeled as points in time in various DSMSs. Therefore, they do not naturally support the notion of validity intervals[1]. Consequently, current DSMSs support a single type of retraction, i.e., full retractions, which occurs at the time of delivery, through the concept of revision tuples [18] or negative tuples [22]. CEDR models the validity interval (application time) of an event and allows retractions to shorten the event validity interval arbitrarily. Some DSMS work describes externally separating the notions of application time and system time [19] in order to resolve out of order delivery. However, current DSMSs internally are sensitive only to a single notion of time. This precludes the possibility of speculative execution and, therefore, the support of a wide spectrum of consistency levels.

[1] An interval can be encoded with a pair of points, but the resulting query formulation will be unintuitive.

To handle the infiniteness of data streams, the *window* operator is utilized in almost every DSMS. There are several notions of the window operator, e.g., time-based windows, tuple-based windows, partitioned windows, windows with a slide, and predicate windows, e.g., [7][21]. Moreover, the window construct can be attached to streams [7] as well as to operators [12]. These windowing approaches commingle the roles of windows as query constructs and windows as state limiters. CEDR separates these concerns. The CEDR *AlterLifetime* operator provides a simple, general technique for handling windowed query processing from a semantic point of view, while memory in CEDR may be explicitly managed separately using the Finalize operator. This is vitally important as windowing (e.g. computing a moving average over one minute) is a query construct with deterministic, well-defined output, while forgetfulness (our Finalize operator) exposes a state/correctness tradeoff.

There have been several approaches to define and implement notions of correctness in DSMSs. In Aurora [12], the output stream is generated on a best effort basis taking into consideration the maximization of a specific measurement for quality of service (QoS). The STREAM system [19] describes the notion of *heartbeats* to limit the blocking time and the memory requirements to generate a highly-consistent output. Nile [14] abides by *view update* semantics as its notion of correctness, that is, a continuous query is semantically equivalent to a *materialized view* on the input streams. More generally, CEDR implements a spectrum of consistency levels where the previous approaches can be mapped into single points inside the consistency domain depicted in Figure [1].

The concept of output retraction is crucial to output consistency. Borealis [18] and Nile [22] generate a revision tuple or a negative tuple, respectively, to retract a previously-generated output in response to a correction event issued by the stream provider. The STREAM system [7] has a similar notion to insertion/deletion tuples through the IStream/DStream operations. However, CEDR supports retractions in response to two scenarios: First, the stream provider issues a correction tuple; Second, CEDR may generate an optimistic output to reduce the blocking time and, later on, CEDR retracts that output and issues a compensation tuple to ensure output correctness.

7. Conclusions

In today's streaming applications, data sources, such as devices, or RFID generators, frequently send their events across unreliable networks. As a result, events frequently arrive at the associated stream processing system out of order. Furthermore, as we discussed in the introduction, some applications generate retractions at the data sources. Due to radically different performance and correctness requirements across different problem domains, systems have been vertically developed to handle a specific set of tradeoffs.

[16] introduced a formal stream model which characterized the full spectrum of behaviors resulting from these stream imperfections. In addition, it introduced a new approach for handling these imperfections based on speculative execution. More specifically, [16] introduced the notion of retractions, which

can be used by operators to remove speculatively produced incorrect output. In addition, it introduced formal semantics independent of data delivery order for operators in the system. Finally, [16] introduced a spectrum of consistency levels characterized by two parameters. The first parameter, maximum blocking time, exposed a tradeoff between the degree of speculation and latency. The second parameter, the maximum time data is remembered before being purged from the system, exposed a tradeoff between state size and correctness. While the second parameter is somewhat known, the first parameter is unique to CEDR. Varying these two parameters produces a spectrum of consistency levels which include the specific tradeoffs built into other systems

In this paper we present the first algorithms for streaming operators that product speculative output. Specifically, we present speculative algorithms for three operators, **`Select`**, **`AlterLifetime`**, and **`Join`**, whose semantics are formally defined in [16], as well as a formal definition and algorithms for a new operator, **`Sum`**. Combined with the two operators introduced in this paper, **`Align`** and **`Finalize`**, the algorithms provided in this paper fully implement the entire spectrum of consistency levels for a rich computational model based on relational algebra. Moreover, these algorithms are provably efficient, and are all either optimal or within a log of optimal for their worst cases. Finally, when state is bounded, as is typically the case for windowed queries over well behaved streams, all algorithms are linear, optimal, and have state complexity of O(1).

8. REFERENCES

[1] Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise system.* Addison Wesley Publishers, 2002.

[2] S. Garfinkel and B. Rosenberg. *RFID: Applications, security, and privacy*. Addison-Wesley Publishers, 2006.

[3] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. *Towards sensor database systems.* In Mobile Data Management, pages 3–14, 2001.

[4] Suman Srinivasan, Haniph Latchman, John Shea, Tan Wong, and Janice Mc-Nair. *Airborne traffic surveillance systems: video surveillance of highway traffic.* In the ACM international workshop on Video surveillance & sensor networks, 2004.

[5] S. Babu, L. Subramanian, and J. Widom. *A data stream management system for network traffic management.* In Proceedings of Workshop on Network-Related Data Management, 2001.

[6] E. Wu, Y. Diao, and S. Rizvi. *High-performance complex event processing over streams.* Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, 2006.

[7] Arvind Arasu, Shivnath Babu, Jennifer Widom. *The CQL continuous query language: semantic foundations and query execution.* VLDB J. 15(2): 121-142 (2006).

[8] D. Zimmer and R. Unland. *On the semantics of complex events in active database management systems*. In ICDE, 392-399, 1999.

[9] S. Madden, M. Shah, J. Hellerstein, and V. Raman, *Continuously Adaptive Continuous Queries over Streams*. In Proceedings of ACM SIGMOD, 2002.

[10] Ron Avnur and Joseph M. Hellerstein. *Eddies: Continuously adaptive query processing.* In SIGMOD Conference, pages 261–272, 2000.

[11] Shivnath Babu and Jennifer Widom. Streamon: *An adaptive engine for stream query processing.* In SIGMOD Conference, pages 931–932, 2004.

[12] D. J. Abadi, D. Carney, U. Cetintemel, et al. *Aurora: A New Model and Architecture for Data Stream Management.* VLDB Journal, 12(2):120–139, 2003.

[13] J. Naughton, D. DeWitt, D. Maier, et al. *The Niagara Internet Query System*. http://www.cs.wisc.edu/niagara.

[14] M. A. Hammad, M. F. Mokbel, M. H. Ali, et al. "*Nile: A Query Processing Engine for Data Streams***.**" In ICDE, 2004.

[15] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M. White: *Cayuga: A General Purpose Event Monitoring System*. In Proceedings of the CIDR Conference on Innovative Data Systems Research, 412-422, 2007.

[16] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. *Consistent Streaming Through Time: A Vision for Event Stream Processing.* . In Proceedings of the CIDR Conference on Innovative Data Systems Research, 412-422, 2007.

[17] M. J. Franklin, et al. *Design considerations for high fan-in systems: The HiFi approach.* In Proceedings of the CIDR Conf on Innovative Data Systems Research, 412-422, 2005.

[18] D. Abadi, et al. *The Design of the Borealis Stream Processing Engine.* In Proceedings of the CIDR Conference on Innovative Data Systems Research, 412-422, 2005.

[19] Arvind Arasu, et al. *STREAM: The Stanford Stream Data Manager.* In Proceedings of the ACM SIGMOD international Conference on Management of Data, 2003.

[20] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, Vladislav Shkapenyuk. *Gigascope: A Stream Database for Network Applications.* SIGMOD Conf, 647-651, 2003.

[21] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. *Exploiting Predicate-Window Semantics over Data Streams.* SIGMOD Record, 35(1):3–8, 2006.

[22] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. 1Mokbel, Walid G. Aref, Ahmed K. Elmagarmid: *Incremental Evaluation of Sliding-Window Queries over Data Streams.* IEEE TKDE 19(1): 57-72, 2007.

[23] Utkarsh Srivastava, Jennifer Widom: *Flexible Time Management in Data Stream Systems.* In PODS, 263-274, 2004.