Safety Guarantee of Continuous Join Queries over Punctuated Data Streams

Hua-Gang Li[1] *
huagang@cs.ucsb.edu

Songting Chen[2]
songting@sv.nec-labs.com

Junichi Tatemura[2]
tatemura@sv.nec-labs.com

Divyakant Agrawal[2]
agrawal@sv.nec-labs.com

K. Selçuk Candan[2]
candan@sv.nec-labs.com

Wang-Pin Hsiung[2]
whsiung@sv.nec-labs.com

[1]Department of Computer Science
University of California Santa Barbara
Santa Barbara, CA 93106

[2]NEC Laboratories America
10080 North Wolfe Road, Suite SW3-350
Cupertino, CA 95014

ABSTRACT

Continuous join queries (CJQ) are needed for correlating data from multiple streams. One fundamental problem for processing such queries is that since the data streams are infinite, this would require the join operator to store infinite states and eventually run out of space. Punctuation semantics has been proposed to specifically address this problem. In particular, punctuations explicitly mark the end of a subset of data and, hence, enable purging of the stored data which will not contribute to any new query results. Given a set of available punctuation schemes, if one can identify that a CJQ still requires unbounded storage, then this query can be flagged as *unsafe* and can be prevented from running. Unfortunately, while punctuation semantics is clearly useful, the mechanisms to identify *if and how* a particular CJQ could benefit from a given set of punctuation schemes are not yet known. In this paper, we provide sufficient and necessary conditions for checking whether a CJQ can be safely executed under a given set of punctuation schemes or not. In particular, we introduce a novel *punctuation graph* to aid the analysis of the safety for a given query. We show that the safety checking problem can be done in polynomial time based on this punctuation graph construct. In addition, various issues and challenges related to the safety checking of CJQs are highlighted.

1. INTRODUCTION

Recent years have witnessed the growth of newly emerging online applications in which data arrives in a *streaming* format and at high speeds. For instance, financial applications process streams of stock market or credit card transactions, telephone call monitoring applications process streams of call-detail records [4], network traffic monitoring applications process streams of network traffic data [5], and sensor network monitoring applications process streams of environmental data gathered by sensors [9]. In such applications, inputs to processing modules take the form of continuous (and potentially infinite) data streams, rather than finite stored data sets. Also, it is quite often that applications require long-running *continuous queries* as opposed to the traditional one-time queries.

One fundamental problem for processing continuous queries is that since data streams are potentially infinite, traditional relational operators, which are well-defined based on finite data, are no longer appropriate. For instance, two highly common operator types are known to be inappropriate for processing infinite data streams: *blocking* operators, such as `groupby`, and *stateful* operators, such as `join` operators. A *blocking* operator may never emit a single result, while a *stateful* operator may require infinite state and eventually run out of memory.

To address these problems, *stream punctuation* semantics has been introduced in the context of data streams [12]. A punctuation is a "predicate" which denotes that no future stream tuples will satisfy this predicate. Thus, based on a given punctuation, stateful and blocking operators may be able to purge data that will no longer contribute to any new results or emit the blocked results, respectively. In short, punctuation semantics breaks the infinite semantics in the streaming context to avoid infinite memory consumption and indefinite blocking. To make the discussion concrete, we borrow the online auction example in [12] as a running example.

EXAMPLE 1. *Here, the* `item` *stream contains items posted by sellers and an item tuple has four attributes, namely,* (`sellerid`, `itemid`, `name`, `initialprice`). *The* `bid` *stream contains the bids posted by buyers and a bid tuple contains three attributes,* (`bidder-id`, `itemid`, `increase`). *An example query in this scenario would be to* "track the difference between the final price and the initial price for each item"*. This can be done by (a) joining the* `item` *stream and* `bid` *stream on their respective* `itemid`*s and then (b) summing up the* `increase` *values for each item seen in the streams.*

Figure 1 illustrates this example. Without any application knowledge, throughout the auction, the system has to keep all incoming tuples from both data streams, since any stored tuple may join with

*The work has been done during the author's internship at NEC Laboratories America.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

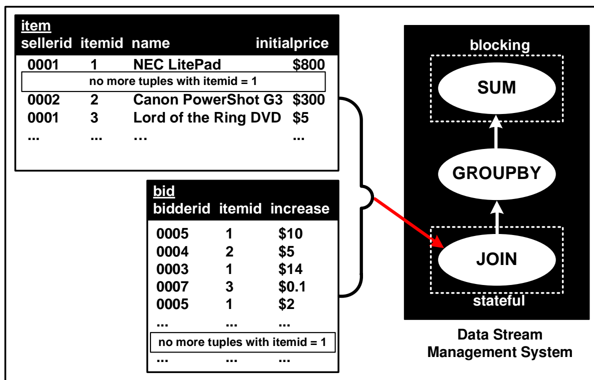

Figure 1: A Punctuation Example from an Online Auction Scenario

a future incoming tuple in the other stream. Thus the query will require infinite join state storage (and the system will eventually break down). Nevertheless, with appropriate punctuations, this *infinite state* problem can be resolved:

- First of all, if each `itemid` is unique in the `item` stream, then each incoming `bid` tuple can join with only one single `item` tuple. Thus, as soon as the corresponding `item` tuple arrives, all the corresponding `bid` tuples can be purged from the system.
- When the auction for one item with `itemid` $= 1$ is closed, then no more bids for the item with `itemid` $= 1$ will be inserted into the `bid` stream. As a consequence, if this information is available (through a punctuation), the `join` operator can purge the `item` tuple with `itemid` $= 1$. Furthermore, the `groupby` operator can now output the result for this item.

The above example shows the utility of punctuations on addressing the infinite state and blocking problems. Unfortunately, it is impossible to guarantee that the punctuation with a specific value, such as `itemid` $= 1$, will always be generated at runtime. The only knowledge we can learn from the application semantics is that there might be punctuations on `itemid` from `bid` stream, which we refer to as *punctuation schemes* (discussed in more detail in Section 2.3). With the knowledge of the available punctuation schemes, it is possible to determine that some continuous queries may still require infinite space no matter what actual punctuations are generated. In the previous example, if the punctuation scheme shows that there are only punctuations on `bidderid` from `bid` stream, then the `item` stream in the above query can *never* be purged. In other words, for some queries, the *infinite state* problem cannot be solved no matter what actual punctuations may come. Such queries are *unsafe* and should not be processed. Obviously, checking safety of continuous queries should be the very first task for a Data Stream Management System (DSMS) [2]. Unfortunately, to our knowledge, there is yet no mechanism that can identify *if and how* a particular continuous query could benefit from the punctuations (or more precisely, *punctuation schemes*) available in the system. Note that although it is impossible to predict which actual punctuations may come at runtime, the safety checking problem under a given punctuation scheme set is very important due to the following reasons:

- First, if the safety checking procedure shows that a query is not safe under a given set of punctuation schemes, then this query should not ever be allowed to be executed, since it can *never* be purged by any actual punctuations. It is important to make such decision at query compile time before such unsafe query consumes all the resources.
- Second, note that there might exist many punctuation schemes defined by the application semantics. It is thus important for the query engine to identify those punctuations that are useful to a particular query. In this way, we can avoid unnecessary processing of the irrelevant punctuations.

The state-of-the-art on the streaming punctuation research mainly addresses the following three issues: (i) semantic modeling of punctuations in [12], in particular, adaptation of the relational operators into stream operators; (ii) generation of useful punctuations, such as the *heartbeats* [11], to ensure correct support of continuous queries on a time basis; and (iii) optimization of the execution behavior in the presence of punctuations, such as the `PJoin` algorithm [6] for optimizing binary joins using punctuations.

In this paper, we focus on the safety checking of the continuous join queries (CJQs) given a set of available punctuation schemes. In particular, we show that while the safety checking is quite straightforward for binary join queries (as shown in the above auction example), it is non-trivial when join queries are defined over more than two data streams (*multi-way join*) and when the punctuation schemes are defined on more than one attribute. The underlying challenges mainly originate from the following two aspects:

- The purge strategy for a multi-way join query is not immediately evident in contrast to that for a binary join query. Especially, it becomes more complicated when punctuation schemes on multiple attributes are available.
- There exist many execution plans for a given multi-way join query. For instance, a multi-way join query can be executed by a single `MJoin` [13] operator, a tree of `MJoin` operators, a tree of binary join operators, or a tree of mixed `MJoin` and binary join operators. However, as we will show in this paper, not *all* these execution plans are safe given the available punctuation schemes. Hence, given a continuous join query, we should be able to decide if this query can be safely executed without having to enumerate all possible execution plans.

Contributions of this Paper. Our contributions in this paper are summarized as follows:

- We formalize the problem of safety checking of continuous join queries under punctuation semantics. This ensures that there is no unlimited memory requirement during query processing.
- We generalize the straightforward purge strategy for the binary join case to the multi-way join case. This generalization, which we call *chained purge strategy*, also serves as the basis for the safety checking of continuous join queries.
- A new graph representation, namely the *punctuation graph*, is proposed to capture the relationship between the punctuation schemes and the join conditions. Based on the punctuation graph construction, we then propose and prove the *necessary* and *sufficient* conditions for checking the safety of continuous join queries.
- We generalize the concept of *punctuation graph* to capture the case when the punctuation schemes involve more than

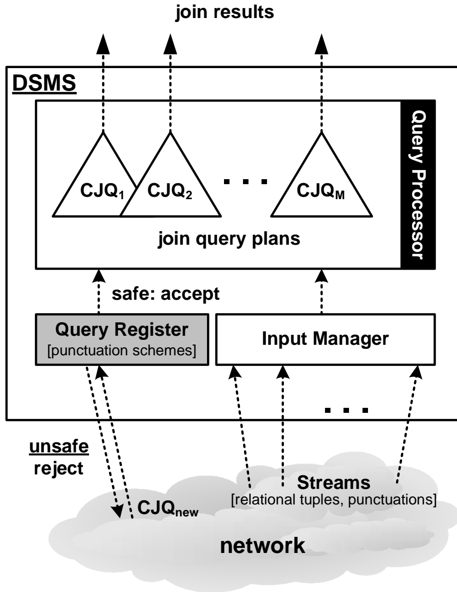

Figure 2: Architecture for Continuous Join Query Processing

one attribute. Finally, a *linear time* safety checking procedure for punctuation schemes on a single attribute, and a *polynomial time* safety checking procedure for punctuation schemes on multiple attributes are presented. Both avoid the exponential enumeration of execution plans of a continuous join query.

- Based on our proposed theoretical framework, we also preliminarily discuss how to choose a safe execution plan for a continuous join query by considering a variety of practical issues.

The rest of the paper is organized as follows. Section 2 presents the notations and problem formulation for safety checking of continuous join queries. Section 3 describes how to purge the join states for both binary join and multi-way join operators. In Section 4, safety checking procedure of continuous join queries is proposed and the correctness proof is presented. We discuss some related issues for safety checking in Section 5. Section 6 reviews the related work, followed by conclusions and future work in Section 7.

2. NOTATIONS AND PROBLEM FORMULATION

In this section, we define the notations used throughout the paper and formalize the safety checking problem of continuous join queries under punctuation semantics.

2.1 System Architecture

Figure 2 depicts an overview of a general DSMS system architecture. The *input manager* accepts and buffers the stream data and punctuations from the application environment. The *query processor* processes the stream data and punctuations for the registered continuous join queries (CJQs). Of course, the system should allow only those CJQs that can be *safely* executed to be registered in the system. In particular, the *query register* records a set of *punctuation schemes* (Section 2.3) which describe the types of punctuations that may be generated for a particular data stream (this information is typically derived from the application semantics). Before registering a continuous join query, the *query register* checks if this query is *safe* from the available *punctuation schemes* (this safety problem will be formalized in Section 2.4). If it is safe, a safe query plan is generated and continuously executed for the incoming stream data (Section 5). Otherwise, the continuous join query is rejected.

2.2 Continuous Join Query and Safety

Continuous Join Query. Each data stream S_i has a relational schema $(A_1^i, \cdots, A_{n_i}^i)$, where each A_j^i is an attribute. A *continuous join query* $CJQ(\Im, \wp)$ is defined over a set of data streams $\Im = \{S_1, \cdots, S_n\}$, where $\wp$ represents a set of join predicates among the data streams.

Each of the join predicates $p \in \wp$ is specified on two data streams S_i and S_j. In this work, we only consider the commonly used *equi-join* predicate, i.e., $A_x^i = A_y^j (1 \leq x \leq n_i, 1 \leq y \leq n_j)$. We also only consider *conjunctive* join predicates between any two data streams. To support other kinds of join predicates and *disjunctive* join predicates remains as future work.

Due to the unbounded nature of data streams, only non-blocking join algorithms are suitable. For instance, a symmetric binary hash join algorithm [14] can be used in the case of binary join operators and a generalized symmetric join algorithm [13] can be employed for `MJoin` operators.

Join State. When executing a continuous join query, inputs of each join operator need to be stored for producing future matches. We refer to the space used for storing the inputs of each join operator as the *join states*. In the case of a hash-based join algorithm, the join states of a join operator refer to the hash tables where the streaming data elements or the intermediate join results are hashed and stored.

We denote $\bowtie^n$ as a join operator with $n (\geq 2)$ inputs (either a binary join operator or an `MJoin` operator), and denote Υ_{S_i} $(i = 1..n)$ as the join states of $\bowtie^n$. Future inputs are denoted as $\Delta\Upsilon_{S_i}$ $(i = 1..n)$. Note that a tuple in Υ_{S_i} needs to be stored as long as it can generate a result with any tuples in the future inputs.

DEFINITION 1. **Purgeability.** *A join state* $\Upsilon_{S_i} (i = 1..n)$ *is purgeable* iff *for any tuple* $t \in \Upsilon_{S_i}$*, there exists a mechanism to determine that* t *will not produce any join results with any new tuples in* $\Delta\Upsilon_{S_j} (j \neq i)$*. A join operator* $\bowtie^n$ *is* purgeable iff *all* n *join states are purgeable.*

Let an execution plan $\Gamma(\Im, \wp)$ of a continuous join query CJQ $(\Im, \wp)$ contain $m (\geq 1)$ join operators; i.e., $\bowtie^{n_1}, \cdots, \bowtie^{n_m}$.

DEFINITION 2. **Safety of an Execution Plan.** *An execution plan* $\Gamma(\Im, \wp)$ *containing* m *join operators* $\bowtie^{n_1}, \cdots, \bowtie^{n_m}$ *is* safe iff *every join operator* $\bowtie^{n_i}$ $(i = 1..m)$ *is purgeable.*

DEFINITION 3. **Safety of a Continuous Join Query.** *A continuous join query* $CJQ(\Im, \wp)$ *is* safe iff *there exists at least one safe execution plan* $\Gamma(\Im, \wp)$.

Note that we need a specific *mechanism* to determine purgeability defined above. When all the data streams are finite as in the conventional database case, the join states can be purged once all the streams are consumed. When we are dealing with *sliding window* type of continuous join queries [2], any tuples in the join states that move out of the time window can be purged. However, when neither of these are applicable, we need a new mechanism to ensure the safety of continuous join queries. Later in this section, we introduce a mechanism to formalize the safety problem using punctuations.

2.3 Punctuation

Punctuation. A *punctuation* P is a predicate on stream elements that must be evaluated to `false` for every element following the punctuation [12]. There are many ways to represent punctuations. We adopt the notations from [12] to represent punctuations as data to allow their easy storage, searching and manipulation. A punctuation for a data stream $S(A_1, \cdots, A_n)$ is formally defined as a set of patterns, one for each attribute $A_i (1 \leq i \leq n)$. As in [12], a pattern can be *wildcard*, denoted as "*". This means that there is no value constraint on a particular attribute for the future stream data. A pattern can also be a constant value, which means that there is an equal-value constraint on a particular attribute for the future stream data. For instance, in the online auction example of Section 1, the punctuation for the `bid` stream which states that no more bids for the item with $\texttt{itemid} = 1$ will arrive can be represented as $(*, \texttt{itemid} = 1, *)$, or simply $(*, 1, *)$.

Punctuation Scheme. In this paper, we introduce the *punctuation scheme* concept to model the application semantics in terms of the formats of punctuations that a data stream S can have. For instance, in the online auction example of Section 1, it only makes sense to have punctuations with constant value patterns on the attribute `itemid` rather than on the attribute `increase` for the `bid` stream.

Formally, a *punctuation scheme* P^S on a data stream $S(A_1, \cdots, A_n)$ is defined as $(P_1^S, \cdots, P_n^S)$. If we may have puncutations with constant value patterns on attribute A_i, then we denote $P_i^S =$ "$+$". In this case, we call attribute A_i *punctuatable* and the actual punctuation p an *instantiation* of its corresponding punctuation scheme P^S. If we can only have punctuations with wildcard patterns on attribute A_i, then we denote $P_i^S =$ "_" and attribute A_i is not punctuatable. In the last auction example, we may have one punctuation scheme on the `bid` stream $(_, +, _, _)$, denoting that we may have punctuations with constant value patterns only on attribute `itemid`.

Note that a punctuation scheme over a data stream may have more than one punctuatable attributes at the same time and the corresponding punctuations have to be instantiated by assigning constant value patterns to all the punctuatable attributes. Moreover, a data stream may have more than one punctuation scheme. The *query register* in Figure 2 contains all the punctuation schemes defined in the DSMS for checking the safety of continuous join queries, referred to as *punctuation scheme set*, denoted by $\Re$.

2.4 Safety Problem Formulation

While punctuations are conceptually useful, as shown in the auction example in Section 1, it is not immediately clear *if and how* a particular continuous join query could benefit from a given set of available punctuations. In this section, we formalize the process through which punctuations affect the safety of a CJQ. We first rephrase Definition 1 in terms of punctuation schemes.

DEFINITION 4. **Purgeability using Punctuations.** *A join state $\Upsilon_{S_i} (i = 1..n)$ of a join operator $\bowtie^n$ is purgeable for a given punctuation scheme set $\Re$* iff *for any tuple $t \in \Upsilon_{S_i}$, there exists a finite set of punctuations $\{P\}$ (with each P being an instantiation of one punctuation scheme in $\Re$) such that we can determine that t will not produce any join results with any new tuples of the join states, $\Delta\Upsilon_{S_j} (j \neq i)$. A join operator $\bowtie^n$ is* purgeable iff *its all n join states are purgeable. An execution plan is safe* iff *all its join operators are purgeable.*

Definition 4 highlights the first problem we tackle in this paper: "*given a punctuation scheme set, how can we determine if the tuples in the join states can be purged?*"

DEFINITION 5. **Safety of a CJQ using Punctuations.** *A continuous join query $CJQ(\Im, \wp)$ is* safe iff *there exists at least one safe execution plan $\Gamma(\Im, \wp)$.*

Definition 5 highlights the second problem we address in this paper. That is, "*given an arbitrary CJQ, how can we decide if there exists a safe plan for executing this query or not?*" As we will show later in Section 4, given the same punctuation scheme set and CJQ, some execution plans are safe while others are not. The challenge is thus to determine the safety of a query without enumerating all possible execution plans, which is obviously very expensive.

Note that there are actually two choices for implementing the purge strategy for continuous join queries. The first choice is a natural one, which is to extend the join operator with purge functionalities [6, 12]. In this case, the query purgeability depends on the shape of the query execution plan. The alternative is to develop a separate purge engine that is independent of any specific query execution plan. In this case, the query purgeability depends only on the query itself. In this work, while we assume using the first model since it is a clear and easy extension to existing query engine, our safety checking results can be applied to both scenarios.

Finally, since a given CJQ can be safely executed in a number of ways under a given punctuation scheme set, we will discuss how to choose a safe execution plan based on various possible objectives.

3. PURGING JOIN STATES USING PUNCTUATIONS

We first address the purgeability of the join states for a given punctuation scheme set. We start with punctuation schemes having only one punctuatable attribute and then we show how to evolve our proposed techniques to handle arbitrary punctuation schemes. Before we can answer *if* a join state can be purged for a given punctuation scheme set, we first need to understand *how to* purge a join state using punctuations. Such a *purge strategy* will serve as the basis for the safety checking procedure in Section 4. In this section, we first study the simple binary join case and generalize the results to the more general `MJoin` case.

3.1 Purging for Binary Join Operator

It is straightforward to determine the required punctuation schemes for a binary join operator's continuous and safe execution. The process was roughly introduced using the online auction example in Section 1. We now formally provide the purge strategy for a binary join operator.

Assume that the two input data streams of a binary join operator $\bowtie^2$ are $S_1(A_1^1, \cdots, A_{n_1}^1)$ and $S_2(A_1^2, \cdots, A_{n_2}^2)$, and the join predicate is $A_i^1 = A_j^2$. In order to purge a tuple $t(a_1, \cdots, a_i, \cdots, a_{n_1})$ in the join state Υ_{S_1} for S_1, we need a punctuation of the form $(*, \cdots, A_j^2 = a_i, \cdots, *)$ from S_2 such that for any new tuples $\Delta\Upsilon_{S_2}$, $t \bowtie \Delta\Upsilon_{S_2}$ must evaluate to ϕ. More formally, in order to purge any tuples in Υ_{S_1}, we need a punctuation scheme P^{S_2} on S_2 with $P_j^S =$ "$+$". A similar situation holds for purging the tuples in the join state Υ_{S_2}.

Note that it is straightforward to support conjunctive join predicates between two input streams. Now we assume that the join predicates are $A_{i_1}^1 = A_{j_1}^2 \wedge \cdots \wedge A_{i_p}^1 = A_{j_p}^2$. A punctuation scheme P^{S_2} from S_2 with at least one $P_k^{S_2} =$ "$+$"$(k = j_1..j_p)$ suffices to purge the join state Υ_{S_1}.

3.2 Purging for MJoin Operator

Prior work [6] on purge strategies of `MJoin` operators only considers the cases where all input streams *share* the same join at-

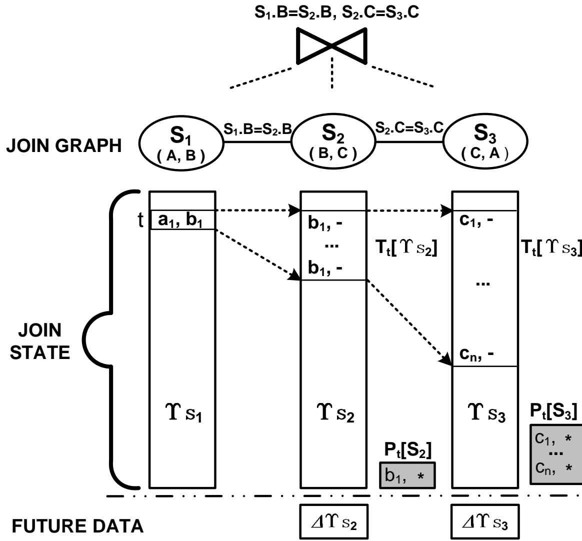

Figure 3: A 3-Way CJQ Executed by an MJoin Operator

tributes in the join predicates. In contrast, in this section, we propose a *chained purge strategy* for the `MJoin` operator under any arbitrary join predicates. We first introduce a notion of *join graph* for a join operator.

DEFINITION 6. **Join Graph.** *The join graph for a join operator $\bowtie^n$ is a connected, undirected, labeled graph $JG(V, E)$. Each vertex $v_i \in V$ represents one input stream S_i for the join operator. Each edge, $e_{ij} \in E$, between any two vertices v_i and v_j represents that there exists a join predicate between S_i and S_j.*

EXAMPLE 2 (MOTIVATING EXAMPLE). *Figure 3 shows an example 3-way join operator with three inputs $S_1(A, B)$, $S_2(B, C)$, $S_3(C, A)$ and two join predicates $S_1.B = S_2.B, S_2.C = S_3.C$. Each vertex in the join graph corresponds to one input. There are two edges, namely, one between S_1 and S_2 and one between S_2 and S_3, denoting the two join predicates.*

Let us now use the above example to provide some intuitions for how to purge the join states for an `MJoin` operator. Assume that the join states for S_1, S_2 and S_3 are $\Upsilon_{S_1}, \Upsilon_{S_2}$ and Υ_{S_3} respectively, as shown in Figure 3. In order to purge a tuple $t(a_1, b_1)$ from Υ_{S_1}, we have to guarantee that it will not generate any new query results with either $\Delta\Upsilon_{S_2}$ or $\Delta\Upsilon_{S_3}$.

First, we consider how to ensure $t \bowtie \Delta\Upsilon_{S_2} = \phi$. Clearly, from the discussions in the last section, we know that we need a punctuation from S_2 as $(b_1, *)$ such that $t \bowtie \Delta\Upsilon_{S_2} = \phi$ always holds. We define the *joinable* tuples in Υ_{S_2} with respect to t as $T_t[\Upsilon_{S_2}] = \Upsilon_{S_2} \ltimes t$ [1]. We further refer to $P_t[S_2]$ as the required punctuations from S_2 for purging tuple t. In this case, $P_t[S_2] = \{(b_1, *)\}$.

Next, we consider how to ensure $t \bowtie (\Upsilon_{S_2} + \Delta\Upsilon_{S_2}) \bowtie \Delta\Upsilon_{S_3} = \phi$. Since $t \bowtie \Delta\Upsilon_{S_2} = \phi$ must hold from previous step, we only need to make sure that $t \bowtie \Upsilon_{S_2} \bowtie \Delta\Upsilon_{S_3} = \phi$. Since $t \bowtie \Upsilon_{S_2} = t \bowtie (\Upsilon_{S_2} \ltimes t) = t \bowtie T_t[\Upsilon_{S_2}]$, we only need to guarantee that $T_t[\Upsilon_{S_2}] \bowtie \Delta\Upsilon_{S_3} = \phi$ always holds. Assume that $\delta_C(T_t[\Upsilon_{S_2}]) = \{c_1, \cdots, c_n\}$ [2]. Again, from the discussion for the binary join case, we need punctuations $(c_1, *), \cdots, (c_n, *)$ to ensure that $T_t[\Upsilon_{S_2}] \bowtie \Delta\Upsilon_{S_3} = \phi$ always holds. The required punctuations are thus $P_t[S_3] = \{(c_1, *), \cdots, (c_n, *)\}$.

The above example shows that there is a *chaining* effect, which results in that streams that are not directly connected with t (in terms of join predicates) still have impact on the purgeability of t.

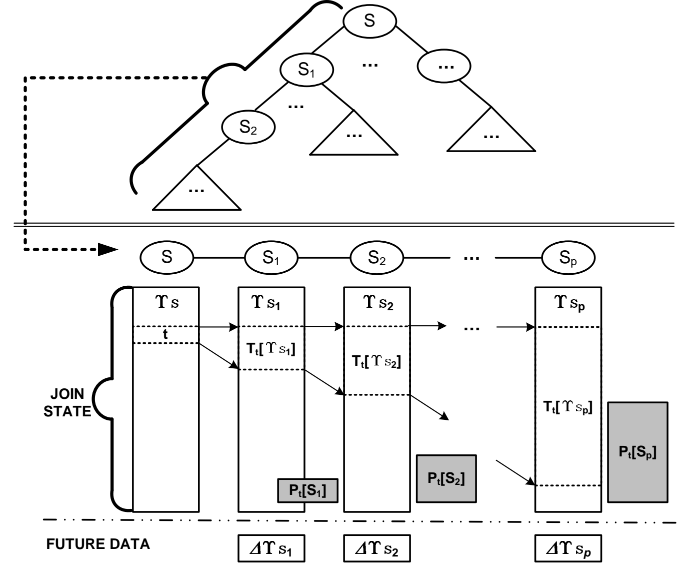

Figure 4: Chained Purge Strategy for MJoin Operator

3.2.1 Chained Purge Strategy

We use the above observation to develop a *chained purge strategy*. First, consider an acyclic join graph. For any node S in the join graph, we can obtain a spanning tree from the join graph rooted at S as shown on the top of Figure 4. Now, consider any root-to-leaf path $S \rightarrow S_1, \cdots, \rightarrow S_p$, with join predicates as $S.A_1 = S_1.A_1$, $S_1.A_2 = S_2.A_2, \cdots, S_{p-1}.A_p = S_p.A_p$ without loss of generality. In order to purge any tuple t in S, we need to ensure that t cannot generate any new query results with $\Delta\Upsilon_{S_1}, \cdots, \Delta\Upsilon_{S_p}$. We derive the required punctuations $P_t[S_i]$ for each S_i in order to purge t below.

Step 1: Obviously, we need punctuations $P_t[S_1]$ with a set of predicates on $S_1.A_1$, whose values come from $\delta_{A_1}(t)$. With $P_t[S_1]$, we guarantee that $t \bowtie \Delta\Upsilon_{S_1} = \phi$ always holds. We then define the *joinable* tuples in Υ_{S_1} with respect to t as $T_t[\Upsilon_{S_1}] = \Upsilon_{S_1} \ltimes t$ for the next step.

Step 2: We need punctuations $P_t[S_2]$ with a set of predicates on $S_2.A_2$, whose values come from $\delta_{A_2}(T_t[\Upsilon_{S_1}])$. With $P_t[S_2]$, we guarantee that $t \bowtie \Upsilon_{S_1} \bowtie \Delta\Upsilon_{S_2} = \phi$ always holds. From previous step, we have $t \bowtie \Delta\Upsilon_{S_1} = \phi$. Together, $t \bowtie (\Upsilon_{S_1} + \Delta\Upsilon_{S_1}) \bowtie \Delta\Upsilon_{S_2} = \phi$ must hold. We can then define the *joinable* tuples in Υ_{S_2} with respect to t as $T_t[\Upsilon_{S_2}] = \Upsilon_{S_2} \ltimes T_t[\Upsilon_{S_1}]$ for the next step.

. . .

Step i: We need punctuations $P_t[S_i]$ with a set of predicates on $S_i.A_i$, whose values come from $\delta_{A_i}(T_t[\Upsilon_{S_{i-1}}])$. With $P_t[S_i]$, we guarantee that $t \bowtie \Upsilon_{S_1} \bowtie \cdots \bowtie \Upsilon_{S_{i-1}} \bowtie \Delta\Upsilon_{S_i}$ must evaluate to ϕ. From all previous steps, we have:

[1] $\ltimes$ denotes semi-join.

[2] δ_C is to select distinct values of attribute C.

$t \bowtie \Delta\Upsilon_{S_1} = \phi$,
$t \bowtie (\Upsilon_{S_1} + \Delta\Upsilon_{S_1}) \bowtie \Delta\Upsilon_{S_2} = \phi$,
$\cdots$
$t \bowtie (\Upsilon_{S_1}+\Delta\Upsilon_{S_1}) \bowtie \cdots \bowtie (\Upsilon_{S_{i-2}}+\Delta\Upsilon_{S_{i-2}}) \bowtie \Delta\Upsilon_{S_{i-1}} = \phi$.

Together, $t \bowtie (\Upsilon_{S_1} + \Delta\Upsilon_{S_1}) \bowtie \cdots \bowtie (\Upsilon_{S_{i-2}} + \Delta\Upsilon_{S_{i-2}}) \bowtie (\Upsilon_{S_{i-1}} + \Delta\Upsilon_{S_{i-1}}) \bowtie \Delta\Upsilon_{S_i} = \phi$ must hold. We then define the *joinable* tuples in Υ_{S_i} with respect to t as $T_t[\Upsilon_{S_i}] = \Upsilon_{S_i} \ltimes T_t[\Upsilon_{S_{i-1}}]$ for the next step.

$\cdots$

Based on the above chained purge strategy, the punctuation scheme P^{S_i} required for each S_i must have $P_i^{S_i} =$ "+", i.e., there are punctuations on $S_i.A_i$.

Finally, note that when the join graph is *cyclic*, there exist *multiple* ways to purge a join state. In Figure 3, if we have an additional join predicate, $S_1.A = S_3.A$, then an alternative way to purge the tuples in Υ_{S_1} would be to first use the punctuations in S_3 on A and then use the punctuations in S_2 on C. In the next section, we will provide a sufficient and necessary condition to check when such a chained purge strategy is applicable under a given set of punctuation schemes for any arbitrary join graph.

4. SAFETY CHECKING PROCEDURE

Based on the chained purge strategy proposed in the last section, we now present the theoretical results for the safety checking problems defined in Definition 4 and 5 in Section 2.4. We progressively present them from the case of punctuation schemes with only one punctuatable attribute to the case of arbitrary punctuation schemes.

4.1 Handling Simple Punctuation Schemes

4.1.1 Purgeability of a Join Operator

Our analysis of the first issue, namely, the purgeability of a join state in Definition 4, is based on the following graph model, called *punctuation graph*, which captures the relationship between join predicates and the corresponding punctuation schemes.

DEFINITION 7. **Punctuation Graph** *Assume that $\bowtie^n$ is a join operator. The punctuation graph of $\bowtie^n$ under a given punctuation scheme set $\Re$ is a* directed *graph, denoted by $PG^{\Re}(\bowtie^n)$. The vertices in $PG^{\Re}(\bowtie^n)$ are the n input streams for $\bowtie^n$, namely, S_1 to S_n. For any join predicate $A_x^i = A_y^j$, if there exists a punctuation scheme in $\Re$ with $P_x^{S_i} =$ "+", then there is a directed edge from S_j to S_i.*

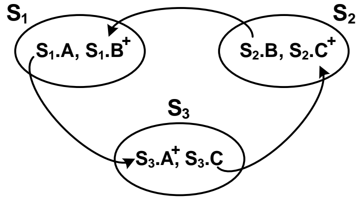

Figure 5: An Example of Punctuation Graph

EXAMPLE 3. *Figure 5 shows an example punctuation graph of a 3-way join operator. The 3-way join operator has three input data streams, namely, S_1, S_2 and S_3. Three join predicates are $S_1.B = S_2.B$, $S_2.C = S_3.C$ and $S_3.A = S_1.A$, respectively. There are also three available punctuation schemes, namely, $(_, +)$ for S_1, $(_, +)$ for S_2 and $(+, _)$ for S_3. The directed edges are created by checking the relationship between the join predicates and the punctuation schemes. For instance, for the join predicate $S_1.B = S_2.B$, since there exists a punctuation scheme of $(_, +)$ from S_1, a directed edge is created from S_2 to S_1. Obviously, such a punctuation graph can be constructed in linear time.*

Based on the notion of punctuation graph, Theorem 1 provides the theoretical result for the first problem in Definition 4.

THEOREM 1. *The join state of an input data stream $S_i (i = 1..n)$ involved in a join operator $\bowtie^n$ is purgeable under a given punctuation scheme set $\Re$* iff *there exists a path from S_i to every other node S_j in the punctuation graph $PG^n(\bowtie^n)$.*

Proof:

"$\Longrightarrow$"

We first prove the following by using contradiction: "If the join state Υ_{S_i} for S_i in a join operator $\bowtie^n$ is purgeable under a given punctuation scheme set $\Re$, then there must exist a path from S_i to every other node S_j in the punctuation graph $PG^n(\bowtie^n)$."

Now assume that S_i cannot reach all other nodes but is still purgeable. There are two classes of nodes in $PG^{\Re}(\bowtie)$ in terms of their relationships to S_i. One class is the nodes that S_i can reach, denoted by R, and the other class is the nodes that S_i cannot reach, denoted by $\bar{R}$. Figure 6 depicts such a classification. We further assume that $S_{X_1}, \cdots, S_{X_p}$ are the input streams in R that have join predicates with the input streams $S_{Y_1}, \cdots, S_{Y_q}$ in $\bar{R}$. $\{S'\}$ are the rest input streams in $\bar{R}$ whose join predicates involve only those input streams in $\bar{R}$, i.e., $\{S'\} = \bar{R} - \{S_{X_1}, \cdots, S_{X_p}\}$.

For a given tuple t in Υ_{S_i}, assume that it can generate some query results and the joinable *tuples (defined in Section 3) in the join state for the input streams in $\bar{R}$ are $T_t[\Upsilon_{S_{Y_1}}]$, $T_t[\Upsilon_{S_{Y_2}}]$, $\cdots$, $T_t[\Upsilon_{S_{Y_q}}]$ and $T_t[\Upsilon_{S'}]$. We now show that it is possible to have new tuples in $\bar{R}$ that are joinable with t and contribute to the new query results. Or in other words, S_i is not purgeable.*

Assume that the input stream S_{Y_j} has attributes $(A_1^{Y_j}, A_2^{Y_j}, \cdots, A_m^{Y_j}, N)$, where $A_1^{Y_j}, A_2^{Y_j}, \cdots, A_m^{Y_j}$ are the join attributes with any of the input stream S_{X_k} in R, and N are the rest attributes that may be joined with the input streams in $\bar{R}$. We further assume one tuple in $T_t[\Upsilon_{S_{Y_j}}]$ as $(a_1^{Y_j}, a_2^{Y_j}, \cdots, a_m^{Y_j}, n)$. Since there cannot be punctuation schemes on any of the attributes $(A_1^{Y_j}, A_2^{Y_j}, \cdots, A_m^{Y_j})$ [3] *and punctuation schemes are limited to one attribute only, it is possible that the values $(a_1^{Y_j}, a_2^{Y_j}, \cdots, a_m^{Y_j})$ will come again.*

We now construct a new tuple for S_{Y_j} as $(a_1^{Y_j}, a_2^{Y_j}, \cdots, a_m^{Y_j}, n_{new})$, where n_{new} are the new values that do not appear in any of the punctuations in $\bar{R}$. Note that we must be able to find such new values n_{new}, since a finite set of punctuations is allowed for purging tuple t. Similar method is used to create new tuples for all $S_{Y_1}, \cdots, S_{Y_q}$. For the input streams $\{S'\}$, we simply create a new tuple with all its attributes being n_{new}.

It is straightforward to show that all these tuples created in $\bar{R}$ are joinable with each other since they all have the same n_{new} values (or already joinable in $T_t[\Upsilon_{Y_j}]$). The resulting tuple is also joinable with the tuples in $T_t[\Upsilon_{S_{x_k}}]$ and subsequently joinable with

[3]Otherwise S_{Y_j} will be reachable through any of the input stream S_{X_k} and thus reachable from S_i.

t. A new result for t will be created. Hence, S_i is not purgeable. This is contradictory to our assumption that S_i is purgeable. In conclusion, S_i must be able to reach every other node.

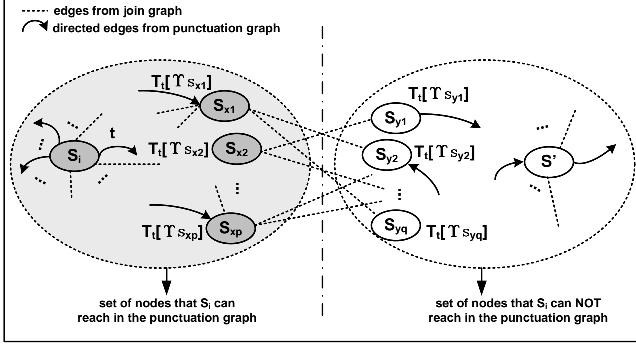

Figure 6: Illustration for the Proof of Theorem 1.

"$\Longleftarrow$"

Next we prove the following: "If, for an input stream S_i involved in a join operator $\bowtie^n$, there exists a path from S_i to every other data stream S_j in the punctuation graph of $\bowtie^n$ under a given punctuation scheme set $\Re$, then the join state of S_i is purgeable under $\Re$."

Since S_i can reach every other node S_j in the punctuation graph $PG^{\Re}(\bowtie^n)$, we can obtain a directed spanning tree T rooted at S_i. Note that there must exist an undirected spanning tree of the same shape in the join graph of $\bowtie^n$. We now prove this claim by applying the chained purge strategy in Section 3.2.1.

Assume that one root-to-leaf path in this spanning tree is $S_i \rightarrow S_1 \rightarrow \cdots \rightarrow S_m$, and the join predicates are $S_i.A_1 = S_1.A_1, \cdots, S_{m-1}.A_m = S_m.A_m$, respectively for each edge. For any tuple t in Υ_{S_i}, starting from S_1, because the attribute $S_1.A_1$ must be punctuatable (from the edge in punctuation graph), we need punctuations $\delta_{A_1}(t)$ to make sure that $\Delta\Upsilon_{S_1}$ will not be joinable with t. The joinable tuples $T_t[\Upsilon_{S_1}]$ in Υ_{S_1} are $\Upsilon_{S_1} \rtimes t$. Clearly, by iteratively applying the chained purge strategy in Section 3.2.1, we can derive a finite set of punctuations for each S_j to purge tuple t. Hence S_i is purgeable. $\square$

From Theorem 1, we immediately get the following corollary which states the necessary and sufficient condition for the purgeability of a join operator. Based on Corollary 1, we can see that the 3-way join operator in Figure 5 is purgeable.

COROLLARY 1. *A join operator $\bowtie^n$ with $S_1, \cdots, S_n$ as input data streams is purgeable under a given punctuation scheme set $\Re$* iff *its punctuation graph under $\Re$, $PG^{\Re}(\bowtie^n)$, is a strongly connected graph.*

4.1.2 Safety Checking of a CJQ

So far, we only investigate the sufficient and necessary condition to guarantee the safety of one join operator, i.e., either a binary join or an `MJoin` operator. As discussed earlier in Section 2, a continuous join query can be processed by an execution plan of either a single `MJoin` operator, a tree of `MJoin` operators, a tree of binary join operators, or a tree of mixed binary join and `MJoin` operators.

The following example shows that even under the same available punctuation schemes and for the same CJQ, some execution plans may be safe while others may NOT. For instance, for the CJQ in Figure 5, while the plan using a single 3-way join operator is safe, *none* of the execution plans based on a tree of binary joins is safe. A sample binary tree is shown in Figure 7: S_1 is joined with S_2 first and the results are then joined with S_3. This execution plan is not safe under the given punctuation scheme set. This is due to the fact that there is no punctuation from S_2 to purge the tuples from S_1 for the lower binary join operator.

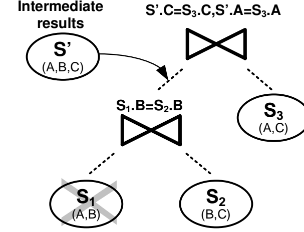

Figure 7: No Safe Execution Plan with Binary Join Operators for CJQ in Figure 5

Theorem 2 provides the theoretical result to the second issue in Definition 5, i.e., if a safe plan exists for a given CJQ under the available punctuation schemes. Here, the punctuation graph of a *CJQ* is defined by assuming the entire query as an `MJoin` operator.

THEOREM 2. *A continuous join query $CJQ(\Im, \wp)$ where $\Im = \{S_1, \cdots, S_n\}$ can be safely executed under a given punctuation scheme set $\Re$ iff its punctuation graph under $\Re$, $PG^{\Re}(CJQ)$, is a strongly connected graph.*

Proof:

"$\Longrightarrow$"

We first prove the following: "If $CJQ(\Im, \wp)$ can be safely executed under a given punctuation schemes $\Re$, then its punctuation graph $PG^{\Re}(CJQ)$ under $\Re$ is a strongly connected graph."

Before we show the correctness of this claim, we first prove the following lemma.

LEMMA 1. *Assume that a purgeable join operator $\bowtie^m$ has m child nodes, $OP_1, \cdots, OP_m$, OP_i can be either another join operator or a data stream. We further assume that $\bowtie^m$ corresponds to a continuous query CJQ, while $OP_1, \cdots, OP_m$ correspond to continuous queries $CJQ_1, \cdots, CJQ_m$. Now CJQ has a strongly connected punctuation graph if all $CJQ_1, \cdots, CJQ_m$ have a strongly connected punctuation graph.*

Proof of Lemma 1: *Assume OP_1 is defined on the raw data streams $(S_1^{i_1}, \cdots, S_{p_1}^{i_1})$, OP_2 is defined on the raw data streams $(S_1^{i_2}, \cdots, S_{p_2}^{i_2})$, $\cdots$, OP_m is defined on the raw data streams $(S_1^{i_m}, \cdots, S_{p_m}^{i_m})$.*

Since the queries correspond to $OP_1, \cdots, OP_m$ have strongly connected punctuation graphs, this means that $(S_1^{i_1}, \cdots, S_{p_1}^{i_1})$ are strongly connected, $(S_1^{i_2}, \cdots, S_{p_2}^{i_2})$ are strongly connected, $\cdots$, $(S_1^{i_m}, \cdots, S_{p_m}^{i_m})$ are strongly connected. To prove CJQ has a strongly connected punctuation graph, we further need to show that all $(S_1^{i_1}, \cdots, S_{p_1}^{i_1}, S_1^{i_2}, \cdots, S_{p_2}^{i_2}, \cdots, S_1^{i_m}, \cdots, S_{p_m}^{i_m})$ are strongly connected.

Since $\bowtie^m$ is purgeable, by Corollary 1, the punctuation graph for $\bowtie^m$ is strongly connected. In other words, $OP_1, \cdots, OP_m$ are strongly connected. Assume that there is one edge from OP_x

to OP_y. Based on Definition 7, this edge means that there exists one join predicate, $S_r^{i_x}.A = S_q^{i_y}.A$, and the attribute $S_q^{i_y}.A$ is punctuatable. Under this join predicate and punctuation scheme, there must also exist an edge from $S_r^{i_x}$ to $S_q^{i_y}$. Given the fact that $(S_1^{i_x}, \cdots, S_{p_x}^{i_x})$ and $(S_1^{i_y}, \cdots, S_{p_y}^{i_y})$ are respectively strongly connected, and $S_r^{i_x}$ can connect to $S_q^{i_y}$, all $(S_1^{i_x}, \cdots, S_{p_x}^{i_x})$ thus can connect to $(S_1^{i_y}, \cdots, S_{p_y}^{i_y})$. This result means that if OP_x can connect to OP_y, then all OP_x involved data streams can also connect to all OP_y involved data streams. By the transitivity property of the edges, all OP_x involved data streams can eventually connect to all other data streams. Finally, since $OP_1, \cdots, OP_m$ are strongly connected, the above is true for the data streams involved in any OP_i. Thus $(S_1^{i_1}, \cdots, S_{p1}^{i_1}, S_1^{i_2}, \cdots, S_{p2}^{i_2}, \cdots, S_1^{i_m}, \cdots, S_{pm}^{i_m})$ are strongly connected - CJQ has a strongly connected punctuation graph. □

We now return to the proof for the sufficient condition of Theorem 2. Since $CJQ(\Im, \wp)$ can be safely executed, according to Definition 3, there must exist at least one safe execution plan $\Gamma(\Im, \wp)$ for it. If $\Gamma(\Im, \wp)$ is the one with an `MJoin` *operator only, then we can deduce the claim is correct based on Corollary 1. Assume that the safe execution plan $\Gamma(\Im, \wp)$ is a generic execution plan tree in which the leaves represent the input streams of $S_1, \cdots, S_n$ and the internal nodes are join operators (either a binary join operator or an* `MJoin` *operator). We now prove the claim using induction based on Lemma 1.*

1) First, all the leaf nodes, i.e., data stream nodes, are trivially true. All the continuous queries correspond to the lowest non-leaf operator nodes in $\Gamma(\Im, \wp)$ must have a strongly-connected punctuation graph, since these operators have only raw data stream input (based on Corollary 1).

2) Now for any purgeable join operator $\bowtie^m$, assume that all the queries correspond to its m child nodes have a strongly-connected punctuation graph, then the query corresponds to $\bowtie^m$ also has a strongly-connected punctuation graph by Lemma 1.

This bottom-up induction based on the plan tree of $\Gamma(\Im, \wp)$ will eventually reach the root of the plan tree. Hence, the entire continuous query has a strongly connected punctuation graph. □

"$\Longleftarrow$"

Next we prove the following: "If the punctuation join graph $PG^{\Re}$ (CJQ) for $CJQ(\Im, \wp)$ under a given punctuation scheme set $\Re$ is a strongly connected graph, then $CJQ(\Im, \wp)$ can be safely executed under $\Re$."

Based on Corollary 1, we know that the execution plan with a single `MJoin` *operator with all $S_1, \cdots, S_n$ as input data streams is safe. Hence, the claim is correct.* □

4.2 Handling Arbitrary Punctuation Schemes

So far, we only discuss the safety checking of CJQs with the case of punctuation schemes having only one punctuatable attribute. In this section, we show how to generalize the proposed technique to handle the case of punctuation schemes with multiple punctuatable attributes.

Let us still take a look at the 3-way join operator as shown in Figure 3 but with the available punctuation scheme set $\Re$ as $\{S_1(_, +)$, $S_2(+, _)$, $S_2(_, +)$, $S_3(+, +)\}$. The join graph and punctuation graph of the 3-way join operator under $\Re$ are shown in Figure 8(a) and (b) respectively. According to Corollary 1, the 3-way join operator is not purgeable since its punctuation graph is not strongly connected. However, the 3-way join operator is actually purgeable in that (i) the join state of S_3 is purgeable according to Theorem 1; (ii) the join state of S_1 is purgeable as can be explained as follows. Assume that $t(a_1, b_1)$ is a tuple from S_1. In order to make sure that t is not joinable with new data coming into S_2, we need a punctuation $(b_1, *)$ from S_2, which can be instantiated by the punctuation scheme $S_2(+, _)$. Furthermore, we assume that t's joinable tuples in S_2 are $(b_1, c_1), \cdots, (b_1, c_m)$. Clearly, if we see punctuations of $(a_1, c_1), \cdots, (a_1, c_m)$ in S_3 instantiated from the punctuation scheme $S_3(+, +)$, together with the punctuation $(b_1, *)$, we can decide that t is not joinable with any new data coming into S_2 and S_3; (iii) following the similar explanation for S_1, the join state of S_2 is also purgeable.

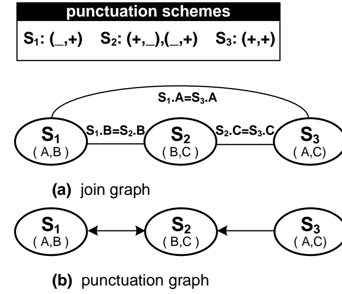

Figure 8: A Motivating Example for Handling Arbitrary Punctuation Schemes

Corollary 1 fails to capture the above case due to the fact that the chained purge strategy does not take the punctuation schemes with more than one punctuatable attribute into account. When we develop the chained purge strategy in Section 3.2.1 for the case of puncutation schemes with only one punctuatable attribute, in step i, in order to make sure $t \bowtie \Upsilon_{S_1} \bowtie \cdots \bowtie \Upsilon_{S_{i-1}} \bowtie \Delta\Upsilon_{S_i} = \phi$, we only need to have the punctuations related to the joinable tuples of t from the previous step. Nevertheless, when we have punctuation schemes with multiple punctuatable attributes, the punctuations related to some/all the join tuples of t from some/all of the previous steps may also suffice to guarantee that $t \bowtie \Upsilon_{S_1} \bowtie \cdots \bowtie \Upsilon_{S_{i-1}} \bowtie \Delta\Upsilon_{S_i} = \phi$. More specifically, let us further take a look at the path from S to S_p as shown in Figure 4. In step i, assume that S_i has $m-1$ extra join predicates with $m-1$ data streams along the path from S to S_{i-1} in which the involved join attributes are $A_{i_1}, \cdots, A_{i_{m-1}}$. To ensure that a tuple t from S is not joinable with any new data from S_i, a punctuation scheme from S_i with the punctuatable attributes (this set is referred to as PA) as a subset of $A_i, A_{i_1}, \cdots, A_{i_{m-1}}$ will suffice to generate a finite number of punctuations of $\delta_{PA}(T_t[\Upsilon_{S_i}])$ to guarantee that, where $T_t[\Upsilon_{S_i}]$ is redefined as $\Upsilon_{S_i} \ltimes (\Upsilon_{S_{i-1}} \bowtie \cdots \bowtie \Upsilon_{S_1} \bowtie t)$. This is to generalize the chained purge strategy to handle the case of punctuation schemes with multiple punctuatable attributes.

Before we proceed to generalize our techniques to handle arbitrary punctuation schemes, we introduce the following three notations: *generalized punctuation graph* (GPG), *reachability in GPG*, and *strong connection in GPG*. The notations are defined in terms of join operator and the companion notations for CJQ can be defined in a similar way.

DEFINITION 8. **Generalized Punctuation Graph** *Assume that $PG^{\Re}(\bowtie^n)$ is the punctuation graph of $\bowtie^n$ under a given punctua-*

tion scheme set $\Re$. The generalized punctuation graph *of* $\bowtie^n$ *under* $\Re$*, referred to as* $GPG^{\Re}$ $(\bowtie^n)$*, is obtained through a generalization procedure on* $PG^{\Re}(\bowtie^n)$ *as follows.*

- *Assume that a data stream* S_i *involved in* $\bowtie^n$ *has a punctuation scheme* P *with* m *punctuatable attributes,* $A_{i_1}, \cdots, A_{i_m}$*, and they are involved as join attributes with data streams* $S_{i_1}, \cdots, S_{i_m}$ *respectively. We create a* generalized node *which covers* $S_{i_1}, \cdots, S_{i_m}$ *and a* generalized *directed edge* $\{S_{i_j}\} \rightarrow S_i$.

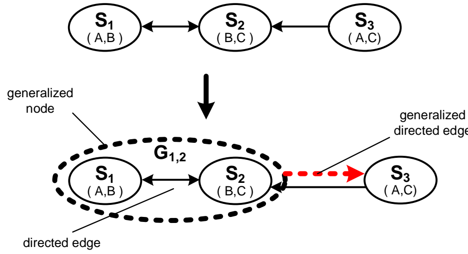

Figure 9: A Generalized Punctuation Graph

EXAMPLE 4. *Figure 9 shows the generalized punctuation graph for the 3-way join operator as shown in Figure 8. Since the punctuation scheme* $(+,+)$ *from* S_3 *has two punctuatable attributes* A *and* C*, and they are involved as join attributes with* S_1 *and* S_2 *respectively, we create a generalized node,* $G_{1,2}$ *and a generalized directed edge from* $G_{1,2}$ *to* S_3 *as shown in Figure 9.*

DEFINITION 9. **Reachability in GPG** *Assume that* $GPG^{\Re}(\bowtie^n)$ *is the generalized punctuation graph of* $\bowtie^n$ *under a given punctuation scheme set* $\Re$*. The reachable node set of a node* S_i *in* $GPG^{\Re}(\bowtie^n)$*,* R_{S_i}*, is defined as follows.*

- *Initially,* R_{S_i} *is set to contain all the nodes which can be reached through the directed edges only.*
- *Repeatedly checking until* R_{S_i} *stops growing: if there exists a generalized directed edge* $\{S_j\} \rightarrow S_x$ *and* $\{S_j\} \subseteq R_{S_i}$*, then* $R_{S_i} = R_{S_i} \cup S_x$.

DEFINITION 10. **Strong Connection in GPG** *Assume that* $GPG^{\Re}$ $(\bowtie^n)$ *is the generalized punctuation graph of* $\bowtie^n$ *under a given punctuation scheme set* $\Re$*. For any pair of nodes* S_i *and* S_j *in* $GPG^{\Re}(\bowtie^n)$*, if* S_i *can reach* S_j *and vice versa, then* $GPG^{\Re}$ $(\bowtie^n)$ *is* strongly connected.

THEOREM 3. *The join state of an input data stream* S_i *involved in a join operator* $\bowtie^n$ *is purgeable under a given punctuation scheme set* $\Re$ iff S_i *can reach every other node* S_j *in* GPG^n $(\bowtie^n)$.

Proof:

"$\Longrightarrow$"

We first prove the following by using contradiction: "If the join state Υ_{S_i} *for* S_i *in a join operator* $\bowtie^n$ *is purgeable under a given punctuation scheme set* $\Re$*, then* S_i *can reach every other node* S_j *in* $GPG^{\Re}(\bowtie^n)$*."*

The proof for the necessary condition is similar to that for Theorem 1 except that we need to point out that when constructing a new tuple for S_{Y_j} *as* $(a_1^{Y_j}, a_2^{Y_j}, \cdots, a_m^{Y_j}, n_{new})$*, Definition 9 can guarantee that there are no punctuation schemes from* S_{Y_j} *whose punctuatable attributes are a subset of* $\{A_1^{Y_j}, A_2^{Y_j}, \cdots, A_m^{Y_j}\}$.[4] *Thus we can have such a tuple to reach a contradiction to our assumption. Note that the proof for the necessary condition of Theorem 1 is not complete when punctuation schemes with multiple punctuatable attributes are available. This is mainly because there is no guarantee that a punctuation scheme will not exist with multiple punctuatable attributes as a subset of* $\{A_1^{Y_j}, A_2^{Y_j}, \cdots, A_m^{Y_j}\}$*, which then prevents us from constructing such a tuple for* S_{Y_j}.

"$\Longleftarrow$"

Next we prove the following: "If an input stream S_i *involved in a join operator* $\bowtie^n$ *can reach every other data stream* S_j *in the generalized punctuation graph of* $\bowtie^n$ *under a given punctuation scheme set* $\Re$*, then the join state of* S_i *is purgeable under* $\Re$*."*

From the definition of reachability in GPG, the reachable node set of S_i*,* R_{S_i}*, initially contains all the nodes which can be reached through the directed edges only. According to the chained purge strategy, we know that there exist a finite number of punctuations in each of the stream (node) in* R_{S_i} *to determine that a tuple* t *from* S_i *will not be able to join with the new data coming into the streams belonging to* R_{S_i}*. When* R_{S_i} *propagates to enclose more reachable nodes through the generalized directed edges, according to the generalized chained purge strategy, we know that there also exist a finite number of punctuations in the newly incorporated reachable node, which can determine that a tuple* t *in* S_i *will not be joinable with any new data coming into the streams belonging to the new* R_{S_i}*. This iterative propagation together with the generalized chained purging strategy will finally guarantee that a tuple* t *from* S_i *will not joinable with any new data coming into all other data streams. Hence,* S_i *is purgeable.* □

Likewise, from Theorem 3, we immediately get the following corollary which states the necessary and sufficient condition for the purgeability of a join operator when handling arbitrary punctuation schemes.

COROLLARY 2. *A join operator* $\bowtie^n$ *is purgeable under a given punctuation scheme set* $\Re$ iff *its generalized punctuation graph* $GPG^{\Re}(\bowtie^n)$ *is strongly connected.*

Now we show the sufficient and necessary condition in Theorem 4 to guarantee the safety of a continuous join query in the presence of arbitrary punctuation schemes.

THEOREM 4. *A continuous join query* $CJQ(\Im, \wp)$ *where* $\Im = \{S_1, \cdots, S_n\}$ *can be safely executed under a given punctuation scheme set* $\Re$ *iff its generalized punctuation graph under* $\Re$*,* $GPG^{\Re}$ (CJQ)*, is strongly connected.*

Proof:

"$\Longrightarrow$"

We first prove the following: "If $CJQ(\Im, \wp)$ *can be safely executed under a given punctuation schemes* $\Re$*, then its generalized punctuation graph* $GPG^{\Re}(CJQ)$ *under* $\Re$ *is strongly connected."*

[4] Otherwise S_{Y_j} will be reachable through any or some of the input streams S_{X_k} and thus reachable from S_i.

Before we show the correctness of this claim, we first prove the following lemma.

LEMMA 2. *Assume that a purgeable join operator $\bowtie^m$ has m child nodes, $OP_1, \cdots, OP_m$, OP_i can be either another join operator or a data stream. We further assume that $\bowtie^m$ corresponds to a continuous query CJQ, while $OP_1, \cdots, OP_m$ correspond to continuous queries $CJQ_1, \cdots, CJQ_m$. CJQ has a strongly connected GPG if all $CJQ_1, \cdots, CJQ_m$ have a strongly connected GPG.*

The proof for this lemma is similar to that for Lemma 1. The only difference is that when we prove the data streams $S_1^{i_x}, \cdots, S_{p_x}^{i_x}$ in OP_x can connect to all the data streams $S_1^{i_y}, \cdots, S_{p_y}^{i_y}$ if OP_x can directly reach OP_y, we should base upon Definition 8. Assume that OP_x can directly reach OP_y. It means (i) there exists a data stream in OP_x which can directly connect to a data stream in OP_y or (ii) there exists a subset of data streams in OP_x together with some data streams from $OP_j (j \neq x)$ which can connect to a data stream in OP_y through a generalized directed edge. Thus, for either case, the data streams in OP_x can connect to all the data streams in OP_y based on the reachability as defined in Definition 9. Then, based on the transitive property, we can prove the GPG for CJQ is also strongly connected. Note that the failure of Theorem 2 to capture the case with arbitrary punctuation schemes in that the proof in Lemma 1 only considers the directed edge between any two data streams, which is restricted by Definition 7.

Based on Lemma 2, the necessary condition of Theorem 4 can be proved similarly as for Theorem 2.

"$\Longleftarrow$"

Next we prove the following: "If the generalized punctuation join graph $GPG^{\Re}(CJQ)$ for $CJQ(\Im, \wp)$ under a given punctuation scheme set $\Re$ is strongly connected, then CJQ $(\Im, \wp)$ can be safely executed under $\Re$."

Based on Corollary 2, we know that the execution plan with a single `MJoin` *operator with all $S_1, \cdots, S_n$ as input data streams is safe. Hence, the claim is correct.* □

4.3 Safety Checking Algorithm

The naïve way to check if the GPG of a continuous join query is strongly connected is to check the reachable node set for each node based on Definition 9, which is obviously expensive in terms of time complexity. Nevertheless, it is not immediately clear to design an efficient algorithm based on the characteristics of a generalized punctuation graph. Thus, we propose to use another practical graph representation, *transformed punctuation graph*, to determine if a continuous join query is safe or not under a given punctuation scheme set. In the following, we will introduce a notation of transformed punctuation graph and show how it is equivalent to the generalized punctuation graph to guarantee the correctness of safety checking procedure.

DEFINITION 11.**Transformed Punctuation Graph** *Assume that $PG^{\Re}(\bowtie^n)$ is the punctuation graph of a join operator $\bowtie^n$ under a punctuation scheme set $\Re$. The transformed punctuation graph of $PG^{\Re}(\bowtie^n)$, referred to as $TPG^{\Re}$ $(\bowtie^n)$ is obtained through the following transformation procedure, which is to iteratively repeat the following steps:*

- *find the strongly connected components;*
- **virtual node construction:** *for each strongly connected component with more than one node, merge them into one new* virtual node *while keeping the structural relationship among the nodes within the strongly connected component;*
- **virtual directed edge construction:** *for any pair of nodes S_i' and S_j' with at least one of them as a virtual node, the join predicate between them is the conjunction of the join predicates, which correspond to the streams covered/represented by S_i' and S_j'. (i)* **directed edge promotion:** *if there exists a directed edge between their covered nodes, then this directed edge is promoted to be as a* virtual directed edge *between S_i' and S_j'. (ii) after the directed edge promotion, if there is still no directed edge from S_i' to S_j' and S_i' is a virtual node, and there exists a punctuation scheme P from one of the streams covered by S_j' (virtual node) or the stream S_j' itself whose punctuatable attributes are a subset of the join attributes from S_j', then add a new virtual directed edge from S_i' to S_j'.*

until the transformed punctuation graph becomes one single virtual node or there does not exist any strongly connected component with more than one node in the transformed punctuation graph.

EXAMPLE 5. *Figure 10 shows the transformed punctuation graph for the PG of the 3-way join operator as shown in Figure 8.*

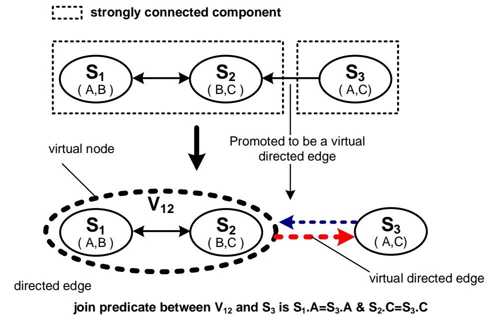

Figure 10: A Transformed Punctuation Graph

THEOREM 5. *Assume that $GPG^{\Re}(CJQ)$ and $TPG^{\Re}(CJQ)$ are the generalized punctuation graph and transformed punctuation graph of a continuous join query $CJQ(\Im, \wp)$ under a given punctuation scheme set $\Re$. $GPG^{\Re}(CJQ)$ is strongly connected* iff *$TPG^{\Re}(CJQ)$ has only one single virtual node.*

Proof:

"$\Longrightarrow$"

We first prove the following by using contradiction: "If $GPG^{\Re}$ (CJQ) is strongly connected, then $TPG^{\Re}(CJQ)$ has only one single virtual node."

Assume $TPG^{\Re}(CJQ)$ has $m(> 1)$ (virtual) nodes in $TPG^{\Re}$ (CJQ), $N_1, \cdots, N_m$ and there is no existence of strongly connected components. Since $GPG^{\Re}$ (CJQ) is strongly connected, every data stream can be purged. Hence, according to the generalized chained purge strategy, every node in $TPG^{\Re}(CJQ)$ has

to connect another node through a directed edge. Therefore, there exist m directed edges for m nodes, which must lead to a cycle in $TPG^{\Re}(CJQ)$. This is contradictory to our assumption. Hence, we prove the above claim.

"$\Longleftarrow$"

Next we prove the following: "If $TPG^{\Re}(CJQ)$ has only one single virtual node, then $GPG^{\Re}(CJQ)$ is strongly connected."

Definition 11 together with the fact that $TPG^{Re}(CJQ)$ is strongly connected implies that every node in $GPG^{\Re}(CJQ)$ can reach any other node. $\Box$

From Theorem 2, we can have a safety checking algorithm based on the transform procedure in Definition 11. It takes linear time to find the strongly connected components and the maximum number of steps for the transformation procedure is $n-1$ where n is the number of data streams involved. Thus, the safety checking algorithm takes polynomial time.

5. DISCUSSION

In this section, we discuss other issues related to the safety checking of *CJQs*. In particular, we focus on the purgeability of the punctuations themselves and on how to choose a safe execution plan among alternatives.

5.1 Punctuation Purgeability

A punctuation helps not only purge the tuples from the current join states, but also purge "future" tuples. Therefore, early removal of the punctuations from the system is potentially hazardous. For example, consider Figure 3: if the punctuation $(b_1, *)$ from the data stream S_2 is simply discarded after purging the tuple (a_1, b_1) in S_1, then any new tuples from S_1 whose attribute B has value b_1 can no longer be purged. Of course, this is not acceptable. On the other hand, storing all the punctuations indefinitely is also not acceptable, as this may lead into infinite memory requirements. Thus, strictly speaking, safety checking of a *CJQ* should involve two kinds of purgeability: *data purgeability* and *punctuation purgeability*. We have already discussed data purgeability in previous sections. In the following, we discuss the punctuation purgeability issue from both theoretical and practical points of view.

A punctuation can be treated as a special tuple and, similar to the normal stream data, punctuations can also be purged by the corresponding punctuations from other streams. For instance, let us re-consider the example in Figure 3. In this example, the punctuation $(*, b_1)$ from S_1 not only helps to remove the tuples in S_2 whose attribute B has value b_1, but also helps to remove the punctuation $(b_1, *)$ from S_2. The reason is that since there will be no more tuples from S_1 whose attribute B has value b_1, $(b_1, *)$ from S_2 no longer needs to be kept.

However, theoretically, purging a normal stream tuple and purging a punctuation are not identical. A normal stream tuple can be purged by punctuations on any of its join attributes, while a punctuation can only be purged by the punctuations on its non-$*$ attributes. For instance, in Figure 5, a tuple (a_1, b_1) from S_1 can be purged by either a punctuation $(b_1, *)$ from S_2 or a punctuation $(*, a_1)$ from S_3, while the punctuation $(*, b_1)$ from S_1 can only be purged by the punctuation $(b_1, *)$ from S_2. Hence, intuitively, we also need punctuations on non-$*$ attributes, clearly rendering punctuation purging very costly in terms of the number of punctuation schemes that need to be supported. We plan to investigate *punctuation purgeability* optimization in our future work.

From a practical stand point, on the other hand, we can argue that strict *punctuation purgeability* is not always necessary. First of all, as was also pointed in [12], basic punctuation semantics (which specifies) that a predicate will be false *forever* is too restrictive and impractical. For instance, such a *forever* semantics would cause serious problems for the reusability of the ID space, if IDs are punctuated. Hence, in order to be practical, punctuations need to have *lifespan*s. As a concrete example from network monitoring applications, a punctuation on both sequence numbers and source IP address may be generated denoting the end of one transmission. According to the TCP RFC [8], the sequence number at a TCP source will cycle approximately every 4.55 hours. This means that such a punctuation has a lifespan for about 4.55 hours. After that, the punctuation needs to expire and can be ignored (i.e., it is implicitly purged). Note that, it may be possible that the same idea of *lifespan* can be applied to purge other types of data as well.

Second, punctuations can also be missed due to the network transmission problems or the application errors. Thus, a background *clean-up* mechanism is mandatory to remove the corresponding *non-purged* data. Since cleaning *missed* non-purged data is much cheaper than cleaning all the data, *data purgeability* alone is sufficient in practice to guarantee the safety of continuous join queries.

5.2 Choosing a Safe Execution Plan

A continuous join query $CJQ(\Im, \wp)$ may be safely executed in numerous ways under a given punctuation scheme set $\Re$. Among all possible safe plans, it is of course desirable to pick one with the minimum cost. In this section, we highlight the potential issues involved in this process and the possible criteria for picking the best execution plan. A full investigation of this optimization problem remains our future work.

Naturally, similar to any traditional query optimization task, this involves *plan enumeration* and *cost estimation* [10]. In this context, *plan enumeration* means the enumeration of possible safe execution plans, while *cost estimation* refers to the estimation of the *cost* for each individual plan.

Plan Enumeration: Given the available punctuation schemes, the number of safe plans is typically much smaller than the number of all possible plans. Thus, rather than first enumerating all possible plans and then checking whether they are safe or not, it is more desirable to generate *only* the safe plans in the first place.

From Section 4, we know that an execution plan is safe *iff* all of its `MJoin` operators (including the binary join operators) are purgeable. By Corollary 1 and 2, each individual `MJoin` operator is purgeable *iff* its (*generalized*) *punctuation graph* is strongly connected. Based on these results, we can conclude that any strongly connected *sub-graph* in the *punctuation graph* for the query could serve as a building block for constructing safe plans. A dynamic programming approach which is similar to the classical system R optimizer [10] can be used to construct the query plan from small strongly connected sub-graphs.

Cost Estimation: Cost estimation when punctuations are considered needs careful consideration. Strictly speaking, punctuations have both costs (in terms of punctuation generation and real-time processing) and benefits (in terms of memory gains, reduced blocking). Therefore, when we talk about cost estimation, we actually refer to a cost/benefit analysis. Since there are many (sometimes conflicting) parameters, such as the *data arrival rate*, *punctuation arrival rate*, and *join selectivities*, involved the goals of the optimization itself may be contradictory: for the simplest example,

consider that one may optimize for memory usage and throughput; but these are not always complementary.

Next, we provide two concrete plan parameter examples and their cost benefit impacts. Consider an `MJoin` operator:

Plan Parameter I: *which alternative punctuation (schemes) to use*. As two extreme cases, consider that we may (a) either choose to use all punctuation schemes available to us, or (b) use only the minimum number of punctuation schemes that will keep the punctuation graph strongly connected. Option (a) is likely to reduce the memory usage for data; but it will increase the memory usage (and the processing cost) for punctuations. Option (b) on the other hand will provide savings in terms of punctuations, but will increase the memory usage for data.

Plan Parameter II: *which runtime purge strategy to use*. A runtime purge strategy [6] can be either *eager* or *lazy*: eager purge strategy processes the punctuations as soon as they arrive, while lazy purge strategy handles punctuations in a batched fashion. As illustrated in [6], different strategies have significantly different impacts on the overall memory usage and system throughput. Therefore, based on the optimization goals, different purge strategies may be applicable.

Adaptive Query Processing: Note that even when an accurate cost model can be developed, the accuracy of the cost model may quickly deteriorate as the system characteristics rapidly change. Unfortunately, such rapid changes and fluctuations are common in a streaming environment. Adaptive query processing has been shown to provide stable performance against fluctuations in data statistics [1, 3]. We believe, it is interesting future direction to study (a) how the changes in the punctuations affect the query performance, (b) how the changes in the data characteristics affect the punctuation plans, and (c) how to implement appropriate adaptation mechanisms.

6. RELATED WORK

Tucker et al. [12] first proposed the streaming punctuation semantics to adapt stateful and blocking operators to the data stream environment. In particular, semantic modeling of punctuations in terms of three kinds of invariants were introduced to specify the proper behavior of query operators in the presence of punctuation. In [6], stream join optimization was discussed by exploiting the punctuation semantics. A new binary join operator `PJoin` was proposed for handling punctuations. In addition, different purge strategies and punctuation propagation strategies were discussed. In [11], a special kind of punctuation, *heartbeats*, is introduced in a Data Stream Management System (DSMS) which relies on time as a basis for windows on streams and for defining a consistent semantics for multiple streams and updatable relations. Heartbeats can support continuous query in an application-defined time domain instead of in a centralized one. None of the above work addresses the safety of continuous join queries under the punctuation semantics.

Another way to limit the space consumption of continuous queries over data streams is to use the window-based semantics, e.g., sliding window. In [3, 7], window joins over data streams were discussed by extending traditional join semantics with stream window semantics. Although window queries are safe in the sense there is no infinite space requirement, exploiting punctuations that are typically available in the data streams can further reduce the memory consumption at runtime.

7. CONCLUSION

In this paper we formalize the problem of safety checking of CJQs under the punctuation semantics. An efficient algorithm for determining the safety of a CJQ is proposed based on a new graph construct, *punctuation graph*. Our proposed technique avoids the exponential enumeration of execution plans when determining if there exists a safe execution plan for a CJQ. Finally, various issues and challenges related to the safety checking of CJQs are highlighted. Our future research work will involve the following aspects: (i) implementation and evaluation on our proposed framework; (ii) supporting disjunctive join predicates for CJQs and arbitrary kinds of punctuation schemes which has more than just one punctuatable attribute; (iii) extend the current safety checking framework or develop a new one for adapting other relational operators to the streaming punctuation semantics; and (iv) supporting the safety checking of an arbitrary SQL-style streaming query.

8. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[4] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *KDD*, pages 9–17, 2000.

[5] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *SIGMOD*, page 623, 2002.

[6] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[7] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[8] Jon Postel, ed. RFC 791: Internet protocol: Darpa internet program protocal specification. September 1981.

[9] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[10] P. G. Selinger, M. M. Morton, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

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

[12] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

[13] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[14] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Int'l Conf. on Databases, Parallel Architectures and Their Applications*, pages 68–77, 1991.