A Heartbeat Mechanism and its Application in Gigascope

Theodore Johnson
AT&T Labs-Research
johnsont@research.att.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Vladislav Shkapenyuk
Rutgers University
vshkap@cs.rutgers.edu

Oliver Spatscheck
AT&T Labs-Research
spatsch@research.att.com

Abstract

Data stream management systems often rely on ordering properties of tuple attributes in order to implement non-blocking operators. However, query operators that work with multiple streams, such as stream merge or join, can often still block if one of the input stream is very slow or bursty. In principle, punctuation and heartbeat mechanisms have been proposed to unblock streaming operators. In practice, it is a challenge to incorporate such mechanisms into a high-performance stream management system that is operational in an industrial application.

In this paper, we introduce a system for punctuation-carrying heartbeat generation that we developed for Gigascope, a high-performance streaming database for network monitoring, that is operationally used within AT&T's IP backbone. We show how heartbeats can be regularly generated by low-level nodes in query execution plans and propagated upward unblocking all streaming operators on its way. Additionally, our heartbeat mechanism can be used for other applications in distributed settings such as detecting node failures, performance monitoring, and query optimization. A performance evaluation using live data feeds shows that our system is capable of working at multiple Gigabit line speeds in a live, industrial deployment and can significantly decrease the query memory utilization.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1 Introduction

A Data Stream Management System (DSMS) evaluates queries over potentially infinite streams of tuples. In order for a DSMS to produce useful output, it must be able to *unblock* operators such as aggregation, join, and union. In general, this unblocking is done by limiting the scope of output tuples that an input tuple can affect. One unblocking mechanism is to define queries over windows of the input stream; this technique is particularly applicable to continuous query systems for monitoring applications [2,3,4,5]. Another technique for localizing input tuple scope is to a timestamp mechanism; this technique is particularly applicable to data reduction applications [8,19].

Our DSMS, Gigascope, requires that some fields of the input data streams be identified as behaving like timestamps. The locality of input tuples is determined by analyzing how the query references the timestamp fields. For example, an aggregation query must have a timestamp field as one of its group-by variables, and a join query must relate timestamp fields of both inputs. Gigascope also has a *merge* operator, which is a union operator that preserves the timestamp property of one of the fields of the input streams.

We have found the timestamp analysis mechanism to be quite effective for unblocking operators as long as all input streams make progress. However, if one of the input streams stalls, operators such as join or merge which combine two streams can stall, possibly leading to a system failure.

Example. Let's consider a concrete example. Gigascope is designed for network monitoring applications. Many of the sites that we monitor have multiple high-speed links (e.g., Gigabit Ethernet) to the Internet. To ensure high reliability, one or more of these links is a *backup* link. If a primary link fails, traffic is automatically diverted to a backup link.

In order to monitor traffic at these installations, we need to monitor all links simultaneously. At a minimum, we need to monitor the merged traffic of a link and its

backup. Since the primary link has gigabit traffic and the backup link has almost no traffic, the merge operator will quickly overflow (i.e., even after optimizations which minimize traffic flow to the merge operator), either running out of buffer space or dropping packets.

The problem we face is that while the presence of tuples carries temporal information, their *absence* does not. A technique that has been proposed in the literature is to use *heartbeats* or *punctuation* [17,20] to unblock operators. However, detailed implementation discussions are lacking.

Our Contributions. We present our implementation of punctuation-carrying heartbeats in the Gigascope DSMS. We first implemented these heartbeats to collect load and liveness information about the operators. Our heartbeats originate at source query operators and propagate throughout the query DAG. We show how timestamp punctuations can be generated at the source query nodes and inferred at every other operator in the query DAG. Finally we show how the punctuated heartbeats can unblock otherwise blocked operators.

In this paper, our focus is on unblocking multi-stream operators such as joins and merges (previous heartbeat work [17] focuses on providing guarantees that tuples arriving to query processor are properly ordered). We demonstrate the need and effectiveness of our punctuation-carrying heartbeats by running experiments with join and merge queries over very high-speed data streams. We find that our punctuation-carrying heartbeat significantly reduces the memory load for join and merge operators with a CPU cost too small to be measured.

Map. The rest of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we provide an overview of two-level (low- and high-) Gigascope DSMS architecture and show how we integrate heartbeats into it. In Section 4, we describe how heartbeat generation is implemented in Gigascope's low-level queries. Section 5 discusses the heartbeat generation and propagation in higher-level queries. We demonstrate how heartbeat mechanism goes beyond its use for operator unblocking by giving its other applications in Section 6. In Section 7, we present our experimental study with Gigascope on live traffic. Conclusions are in Section 8.

2 Related Work

A heartbeat is a very widely used mechanism in distributed systems to achieve fault tolerance. The most common implementations have remote nodes send periodic hearbeat messages to inform other nodes that they are alive. If no hearbeat is received for certain amount of time, the node is declared dead. Recent research projects in distributed data stream management systems (Aurora+, Medusa, and Borealis) [1,6] also use hearbeats to detect remote node failures.

Stream punctuation [13,14,20] has been proposed as a technique to unblock operators by embedding special marks in the stream that indicate the end of a subset of the data. This mechanism is very generic and allows punctuation to carry arbitrary information that might be helpful to operators (e.g. all future tuples will have the values of the attribute in certain range). However, this work on punctuated streams does not describe how data sources are going to generate the punctuations. It is also not clear how to integrate such a mechanism into high-performance streaming database that needs to process data at line speeds.

The heartbeat mechanism described in [17] is designed to enforce a guarantee that all the tuples are ordered by a timestamp before they are sent to the query processor. This approach assumes that the DSMS includes a special *input manager* that buffers tuples arriving from multiple streams to provide such a guarantee. They focus on eliminating out-of-orderness in input streams, which is different from our problem of unblocking multi-stream operators. Gigascope's punctuation-carrying heartbeats that we present here are not restricted to a single system or application time, and are designed for large number of protocol and application-level timestamps and sequence numbers characteristic of network streams.

3 Integrating Hearbeats in Gigascope

Gigascope [7,8,9] is a high-performance streaming database designed for monitoring of the networks with high-rate data streams. In this section, we discuss some relevant aspects of the Gigascope architecture and how we integrate heartbeats into Gigascope runtime and code-generation system.

3.1 Gigascope Architecture

Gigascope is designed for monitoring very high speed data streams using inexpensive processors. To accomplish this goal, Gigascope uses an architecture which is optimized for its particular application. For one, Gigascope is a stream-only database—it does not support stored relations or continuous queries. This restriction greatly simplifies and streamlines the implementation. However, since there are no continuous queries (as implemented in, e.g., [18]) there are no explicit query evaluation windows, which are necessary to unblock operators such as aggregation and join. Instead, attributes in streams can be labeled with a "timestampness", such as *monotone increasing*. The query planner uses this information to determine how (and whether) a blocking operator can be unblocked.

In an aggregation query, at least one of the group-by attributes must have a timestampness, say monotone increasing. When this attribute changes in value, all existing groups and their aggregates are flushed to the operator's output (similar to the tumble operator [4]). The values of the group-by attributes with timestampness thus define *epochs* in which aggregation occurs, with a flush at the end of each epoch.

For an example, suppose that `time` is labeled as monotone increasing in the `TCP` stream. Then in the following query:

```
SELECT tb, srcIP, count(*) from TCP
GROUP BY time/60 as tb, srcIP
```

The `tb` group-by variable is inferred to be monotone increasing also. This query counts the packets from each source IP address during 60 second epochs.

A merge operator performs a union of two streams R and S in a way that preserves timestamps. R and S must have the same schema, and both must have a timestamp field, say t, on which to merge. If tuples on one stream, say R, have a larger value of t than those in S, then the tuples from R are buffered until the S tuples catch up.

A join query on streams R and S must contain a join predicate such as `R.tr=S.ts` or `R.tr/2=S.ts+1`: that is, one which relates a timestamp field from R to one in S. The input streams are buffered (in a manner similar to that done for merge) to ensure that the streams match up on the timestamp predicate.

- Use *early data reduction* to handle very high speed data streams.
- *Low-level queries* perform initial fast selection and aggregation on high speed stream.
- Fixed-size buffers at the low level
- Finalize aggregation in post processing.

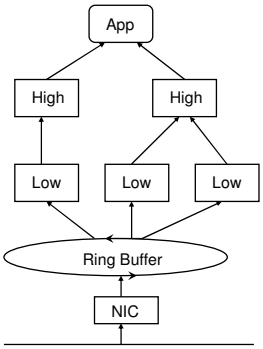

Figure 1: Gigascope architecture.

Another aspect of Gigascope's specialization is its *two-level query architecture*, where the low level is used for data reduction and the high level performs more complex processing. This approach is employed for keeping up with high streaming rates in a controlled way. High speed data streams from, e.g. a Network Interface Card (NIC), are placed in a large ring buffer. These streams are called source streams to distinguish them from data streams created by queries. The data volumes of these source streams are far too large to provide a copy to each query on the stream. Instead, the queries are shipped to the streams. If a query Q is to be executed over source stream S, then Gigascope creates a subquery q which directly accesses S, and transforms Q into Q0 which is executed over the output from q. In general, one subquery is created for every table variable which aliases a source stream, for every query in the current query set. The subqueries read directly from the ring buffer. Since their output streams are much smaller than the source stream, the two-level architecture greatly reduces the amount of copying (simple queries can be evaluated directly on a source stream).

The subqueries (which are called "*LFTAs*", or low-level queries, in Gigascope) are intended to be fast, lightweight data reduction queries. By deferring expensive processing (expensive functions and predicates, joins, large scale aggregation), the high volume source stream is quickly processed, minimizing buffer requirements. The expensive processing is performed on the output of the low level queries, but this data volume is smaller and easily buffered. Depending on the capabilities of the NIC, we can push some or all of the subquery processing into the NIC itself. To ensure that aggregation is fast, the low-level aggregation operator uses a fixed-size hash table for maintaining the different groups of a GROUP BY. If a hash table collision occurs, the existing group and its aggregate are ejected (as a tuple), and the new group uses the old group's slot. That is, Gigascope computes a partial aggregate at the low level which is completed at a higher level. The query decomposition of an aggregate query Q is similar to that of subaggregates and superaggregates in data cube computations [10].

A DSMS has many aspects of a real-time system: if the system cannot keep up with the offered load, it will drop tuples. We implemented traffic-shaping policies in some of the Gigascope operators to spread out processing load over time and thus improve ability to schedule. In particular, the aggregation operator uses a *slow flush* to emit tuples when the aggregation epoch changes. One output tuple is emitted for every input tuple which arrives, until all finished groups have been output (or the epoch changes again, in which case all old groups are flushed immediately).

3.2 Using hearbeats to unblock streaming operators

We initially designed a heartbeat mechanism to collect the runtime statistics about operator load and to detect node failures when system is used in distributed settings. Gigascope heartbeats are special messages that are regularly produced by low-level query operators and propagated throughout the query DAG. Since heartbeat messages are propagated using the regular tuple routing mechanism, they incur the same queuing delays as regular tuples and can give a good indication of the system bottlenecks and overloaded nodes. Collecting traces of the heartbeats propagating through query execution DAG

gave us a valuable tool in system performance monitoring. Another benefit of having regular flow of heartbeat messages through active operators is the ease of detecting failed nodes.

We approached the problem of unblocking streaming operators that take multiple inputs by implementing a stream punctuation mechanism which injects special temporal update tuples into operator output streams. The purpose of temporal update tuples is to inform the receiving operators about the end of a subset of a data (typically the end of the time window or epoch on which streaming operators such as aggregations, stream merge and joins operate). Our first implementation of stream punctuations used on-demand generation of temporal updates tuples. In this approach blocked operators explicitly request the temporal update tuple from their input nodes. We found that on-demand generation of stream punctuations led to unnecessary complexity in both Gigascope runtime and code generation system. After taking a closer look at Gigascope heartbeat mechanism we realized that heartbeats regularly propagating through query execution DAG provide a perfect vehicle for carrying the temporal update tuples. The constant flow of punctuation-carrying heartbeats ensures that stalled merge and join operators will be unblocked in timely manner.

Temporal update tuples generated by streaming operator have identical schema to regular tuple, but they also have a few important distinctions. All the tuple attributes that are marked as temporal in operator's output schema are initialized with values that are guaranteed not to violate the ordering properties. For example, if attribute *Timebucket* is marked as *temporal increasing*, and operator receives a temporal update tuple with value *Timebucket* = *t*, all future tuple are guaranteed to have *Timebucket* >= *t.* All the non-temporal attributes in stream schema are left uninitialized and are ignored by receiving operators. One simple and very conservative scheme for generating such temporal tuples is to always emit the previously produced tuple (cast as a temporal update tuple). However, such mechanism would be useless as heartbeats will not provide any new information to streaming operators. Our goal is to build a system that will be very aggressive in generating values of temporal attributes and try to set them to highest possible value it can safely guarantee (or lowest value in case of *temporal decreasing* attributes). We will describe our algorithms for generating the values of temporal attributes in sections 4 and 5.

4 Heartbeat Generation at LFTA Level

Heartbeat generation in Gigascope is initiated by low-level operators (LFTAs) regularly injecting the heartbeat messages carrying temporal update tuples into their output streams. In this section, we a give brief overview of low-level streaming operators used in Gigascope and describe the algorithms generation of temporal update tuples.

4.1 Low-level streaming operators in Gigascope

Gigascope's low-level streaming operators (LFTAs) read data directly from *source data streams* (e.g., packets sniffed from a network interface). Their main purpose is to maximally reduce the amount of data in a stream using filtering, projection and aggregation before it is passed to higher-level execution nodes (requiring a memory copy). Input tuples, typically in the form of networks packets, are read directly from NIC's ring buffer. To avoid overflowing this high input rate buffer, it is essential that the processing of input tuples be as fast as possible. The only two types of streaming operators used in LFTA nodes are selection and aggregation. Gigascope uses a large number of optimizations to cut down the LFTA processing costs. Low-level operators are compiled into C code that is linked directly to runtime library, to avoid expensive runtime query interpretation. Gigascope also performs a limited form of multi-query optimization through a *prefilter*, discussed below.

The normal mode of operations of the LFTA node in Gigascope is to block, waiting for new tuples to be posted to a NIC's ring buffer. Once a tuple is posted in the buffer, the runtime system invokes operator's *Accept_Tuple()* function to process it. In order to make sure that operators regularly produce the heartbeats even in the absence of incoming packets, the runtime system periodically interrupts the LFTA's wait and requests for them to emit a punctuation-carrying heartbeat.

Every low-level operator maintains the necessary state required to correctly generate temporal update tuples. This state always includes the last seen values of all the temporal attributes referenced in operator's select clause, in addition to other operator-specific states. These values are used by the operator to infer the values of the temporal attributes for temporal update tuples. The example of such an inference is given in the following aggregation query:

```
SELECT tb, srcIP, count(*) from TCP
GROUP BY time/60 as tb, srcIP
```

If according to LFTA's internal state the last seen value of 'time' attribute was X, it will use the inference rules to generate a 'tb' value for temporal update tuple to be equal to X/60.

4.2 Effects of prefilters

Preliminary filtering is a form of multiple query optimization employed by Gigascope to avoid the cost of invoking operators on tuples which are certain to fail selection predicates. Even though this technique frequently leads to significant performance gains, it presents a problem for our heartbeat generation system. Consider a scenario in which an arriving tuple has a value of the temporal attributes that would advance the time

window used by higher-level aggregation, merge or join operator. If the tuple failed the prefilter test, it will never be delivered to LFTA operators and they would not be aware that the time window in fact advanced.

In order to avoid losing valuable temporal information, we augment the prefilter to save the values of all the temporary attributes used by the queries that share the prefilter. These saved values are made available to all LFTA nodes for use in heartbeat generation.

4.3 Heartbeats in selection LFTAs

Low-level selection operators in Gigascope perform selection, projection, and transformation on packets arriving from a source data stream. The normal tuple processing flow for this operator is to unpack the values of the fields referenced in the query predicate and check if the predicate is satisfied. If so, the output tuple is generated according to the projection list in query select clause. There are a small number of changes that need to be made to the normal tuple processing flow in order to enable heartbeat generation:

1) Modify operator's *Accept_Tuple()* function to save the values of all temporal attributes referenced in query Select clause.
2) Whenever operator receives a regular request to generate a temporal update tuple, use the maximum of the saved value of temporal attributes and a value saved by prefilter to infer the value of the temporal update tuple.

It is important to note that the values of the temporal attributes are saved in *Accept_Tuple()* regardless of whether tuple satisfies the operator's predicate or not. The generation of attribute values for temporal update tuples is done using the value inference scheme outlined earlier in Section 4.1.

4.4 Heartbeats in aggregation LFTAs

Gigascope's low-level aggregation queries implement group by and aggregation functionality using small direct-mapped hash table. Whenever a collision in a hash-table occurs, the ejected tuple is sent to output stream; as a result, the output stream can have multiple tuples for the same group. To ensure that the aggregation query always generates the correct output, a low-level query is paired with high-level aggregation node that completes the aggregation of partial results produced by LFTA.

Whenever the incoming tuple advances the epoch, the aggregation operator closes all the aggregates maintained in the hash table and flushes them to the output stream. If the number of groups accumulated during an epoch is very large, the flush puts a large load on a stream manager and can potentially lead to overflow of system buffers. To avoid this effect Gigascope uses a traffic-shaping technique known as *slow flush*. Instead of putting tuples directly into output stream, it gradually emits them as new tuples arrive from the input. This property has a significant effect on generating the values of temporal attributes in heartbeat tuples. Using the largest observed values of temporal attributes may violate the stream ordering properties because some tuples with smaller attribute values remain unflushed.

Similar to selection operator, aggregation nodes save the last seen values of temporal attributes in the input stream and use the value inference to generate temporal update tuples. In addition to the state common to all operators, it also maintains the value of temporal attributes of the last tuple it flushed to the output stream. Whenever a request to produce a heartbeat is received, the following formula is used:

```
if we have unflushed tuples :
        use the value of last flushed tuple
else:
        use maximum of the saved value of temporal
        attributes and the value saved by the prefilter
```

This method guarantees that heartbeat tuples injected into operator's output stream do not violate temporal attribute ordering properties.

4.5 Infering values of temporal attributes based on system time

The heartbeat generation scheme that Gigascope uses in LFTAs works well when each of the monitored links has some amount of traffic. However, the situation becomes more complicated when one of the monitored network cards does not observe any tuple in a long time. In the absence of incoming tuples, the streaming operators will not be able to advance the values of temporal attributes and will conservatively produce heartbeats based on previously observed input. Since most of the temporal attributes in typical network queries are time-based and can be easily correlated with system clock, naturally, Gigascope has the ability to advance the values of temporal attributes based on a system clock.

When advancing temporal attributes using this method, one must however be careful about the skew between the system clock and the timestamps assigned by network interfaces. One source of the skew is the buffering in packet capture library (pcap) library that can keep already timestamped tuples from being delivered to LFTA nodes. In the presence of low-rate stream, buffering can lead to scenarios where the timestamps of the tuples received by LFTA fall significantly behind a system clock. As part of setting up Gigascope, the administrator needs to specify the maximum skew between host system clock and each of the monitored network interfaces. The heartbeat generation system uses the skew information to automatically advance the time-

based temporal attributes. Future tuples that violate the skew specification are discarded by receiving LFTAs.

5 Heartbeat Propagation at HFTA Level

A streaming operator in high-level query nodes (HFTA) emits temporal update tuples whenever it receives a heartbeat from one of its source stream. In this section, we give an overview of high-level query nodes and the streaming operators they use as well as algorithms for heartbeat generation and propagation by different streaming operators.

5.1 High-level query nodes in Gigascope

An important characteristic of the Gigascope architecture is a two-level approach to query execution. Low-level subqueries (LFTAs) executing directly within Gigascope runtime are responsible for early data reduction, while more complicated processing involving expensive predicates or complex operators is performed in high-level query nodes (HFTAs). Even though from application perspective LFTAs and HFTAs are indistinguishable, there are significant differences in their capabilities. High-level nodes are not restricted to running single streaming operator (the way LFTAs are) and can implement arbitrarily complex query execution plans. Currently Gigascope supports selection, multiple types of aggregation, stream merge, and inner and outer join operators. HFTAs can receive data from multiple different streams produced by LFTAs and other running HFTAs.

The normal mode of execution of an HFTA node in Gigascope is to block, waiting for new tuples to arrive from one of its input streams. After determining which operators in the query execution tree are subscribed to that input stream, the runtime system invokes operator's *Accept_Tuple()* function to process the incoming tuples. If the processing of the tuple forces the operator to produce some output tuples, they are routed to the appropriate parent operator in query execution plan. In addition to the regular tuples arriving from one of its input stream, an HFTA regularly receives temporal update tuples produced by LFTAs or other HFTAs. We augmented the implementation of all streaming operators to correctly interpret temporal update tuples and use them to unblock themselves. We will describe the changes that we made in subsequent sections dedicated to different types of operators.

Similar to low-level operators described earlier, high-level operators residing in an HFTA maintain the necessary state required to generate temporal update tuples. Normally the state includes the last seen values of all relevant temporal attributes for each of the operator's input streams. High-level operators use these values in addition to operator-specific state to infer the values of the attributes of temporal update tuples.

5.2 Heartbeats in selection operator

Heartbeat generation in selection operator is largely identical to the scheme used selection LFTAs discussed earlier. The difference lies in the fact that operator can receive temporal update tuples in addition to regular data tuples. Whenever a temporal update tuples is received, operator updates the saved values of all temporal attributes referenced in query Select clause and generates a new temporal update tuple based on a saved state. The rest of the normal tuple processing is bypassed. The generation of attribute values for temporal update tuples is done using the value inference scheme outlined earlier in Section 4.1.

5.3 Heartbeats in aggregation and sampling

The high-level aggregation operator in Gigascope is a non-blocking operator that aggregates the data within a time window (epoch) defined by values of temporal groupby attributes. In contrast with low-level aggregation queries that use direct-mapped hash-table and can emit multiple partial aggregates for the same group, high-level aggregates are required to keep all the groups and corresponding aggregates till the end of epoch before flushing them to output stream. To deal with the increased danger of overflowing system buffers by flushing huge amounts of data at the end of the epoch, aggregation operators rely on the slow flush mechanism that we described earlier.

We made a small number of modifications to Gigascope aggregation operator to enable the generation of the punctuation-carrying heartbeats. These modifications mostly mimic the changes required to implement heartbeats in low-level aggregation queries. The operator maintains the last seen values of all relevant temporal attributes, updating them whenever a new tuple (regular or temporal) arrives. In addition to this state, the operator also maintains the values of temporal attributes of the last flushed tuple (for correctness in the presence of slow flush). These values are combined to infer the attributes of temporal update tuple using the formula from Section 4.4.

In addition to traditional stream aggregation operators, Gigascope also supports more complex aggregation operators – such as the stream sampling operator [21]. However, in all respects related to processing of temporal tuples and heartbeat generation, these operators behave identically to plain aggregation operator and share all the heartbeat-related code.

5.4 Heartbeats in stream merge oprator

A merge operator in Gigascope performs a union of two streams R and S in a way that preserves the ordering properties of the temporal attributes. R and S must have the same schema, and both must have a temporal field, say *t*, on which to merge. Note that *t* is the only attribute that preserves the temporal properties in the merge output

schema. The operator maintains the smallest values R_{MIN} and S_{MIN} of the timestamp observed on each of the input streams. If tuples on one stream, say R, have a values of *t* larger then S_{MIN}, then the tuples from R are buffered until the S tuples catch up. Note that the values of R_{MIN} and S_{MIN} are updated whenever a new tuple (regular or temporal) arrives from a stream that has no buffered tuples. Whenever the operator is asked to generate the temporal update tuple, it can trivially generate it by setting the value of *t* to MIN(R_{MIN}, S_{MIN}).

5.5 Hearbeats in join oprator

GSQL queries that join two data streams R and S must contain a predicate that relates a timestamp from R to one in S (e.g. R.tr = 2 * S.ts). This requirement is critical for implementing the join using bounded amount of memory without relying on sliding windows. Gigascope implementation of join operator supports inner as well as left, right and full outer equi-joins. Similar to merge operator, join maintains a minimum timestamps R_{MIN} and S_{MIN} and buffers input streams to ensure they match up on the timestamp predicate. Note that timestamp in GSQL may include a number of temporal attributes, so R_{MIN} and S_{MIN} could be a composite structure storing minimum values of all attributes that constitute a timestamp. Again the value of the attributes in temporal updates tuples are generated using the MIN(R_{MIN}, S_{MIN}) formula.

6 Other heartbeat applications

Our initial goal in implementing heartbeat mechanism for Gigascope was to collect the statistics about the load on query nodes when system is used in distributed settings. Once the mechanism was implemented we discovered that heartbeat infrastructure can be used for variety of other tasks. In addition to carrying stream punctuations (which is the main focus of this paper) and statistics collection, we are currently applying heartbeats to fault tolerance, query performance analysis, distributed query optimization. In this section we give brief overview of different applications that rely on Gigascope's heartbeats.

6.1 Fault tolerance

A heartbeat is a widely used mechanism in distributed systems to detect node failures. Traditional implementations require every remote node to periodically send heartbeat messages to a resource manager to indicate that the node is still alive. In our Gigascope implementation, we use a slightly different scheme in which heartbeats are periodically generated by low-level queries and propagated upward through the query execution DAG. A constant flow of heartbeat tuples through the system provides an easy way for a running query to identify that a node running one of its subqueries no longer responding. If a subquery does not produce a heartbeat for some specified amount of time, it is declared to be failed and a recovery procedure is initiated. Usually the recovery involves moving an instance of the failed query to another machine.

6.2 System performance analysis

Gigascope relies on the regular tuple routing mechanism to propagate the heartbeat messages from the low-level queries up to top-level nodes that applications subscribe to. As a result, heartbeats are subject to the same queuing delays that regular tuples incur and can be used to identify backlogged nodes and system bottlenecks. Every heartbeat message emitted by an LFTA is timestamped and contains an identifier of the producing query. In addition to this information, every heartbeat is assigned a special trace identifier (*trace_id*). As the messages propagate upwards to higher level nodes, they attach their own identifiers along with a timestamps corresponding to the time they received a heartbeat. When a heartbeat message finally reaches a top-level query node, it has a full trace of all the operators it visited on its way along with the delays it incurred in each of the operator's queues. Gigascope administrators can use these heartbeat traces to identify system performance problems that are otherwise very difficult to detect.

6.3 Distributed query optimization

The Gigascope team is currently working on automated tools that will be able to utilize the statistics collected using the heartbeat mechanism and dynamically re-optimize the query execution plans to eliminate identified bottlenecks. In addition to detailed traces described in previous section, we plan to make heartbeats carry other operator statistics such as predicate selectivities, data arrival rates and tuple processing costs.

7 Performance evaluation

In this section we present our experiments with the Gigascope heartbeat mechanism. These experiments were conducted on a live network feed from a data center tap. All our queries monitor the set of 3 network interfaces, two high-speed DAG4.3GE Gigabit Ethernet interfaces (*main1* and *main2*) which see the main bulk of the traffic and one control 100Mbit interface (*control*). Both Gigabit interfaces receive approximately 100,000 packets per second (about 400 Mbits/sec). Our primary focus is to be able to unblock streaming operators that combine the streams from both high-rate main links and low-rate backup links. Since the control interface has very small amount of traffic, its behaviour is a representative of the behaviour of backup interfaces. All experiments were

conducted on dual processor 2.8 GHz P4 server with 4 GB of RAM running FreeBSD 4.10.

7.1 Unblocking stream merge using hearbeats

We evaluated the effect that punctuation-carrying heartbeats have on memory usage of running queries that use the stream merge operator. For this experiment we used the following GSQL query:

```
SELECT tb,protocol,srcIP,destIP,
      srcPort, destPort, count(*)
FFROM DataProtocol
GROUP BY time/10 as tb, protocol,
      srcIP, destIP, srcPort, destPort
```

The query computes the number of packets observed in different flows in 10 second time buckets. Since the query is executed on a machine with 3 network interfaces, the Gigascope query planner automatically inserts stream merge operators into query plans to combine the stream from different interfaces. The resulting query plan is shown in Figure 3.

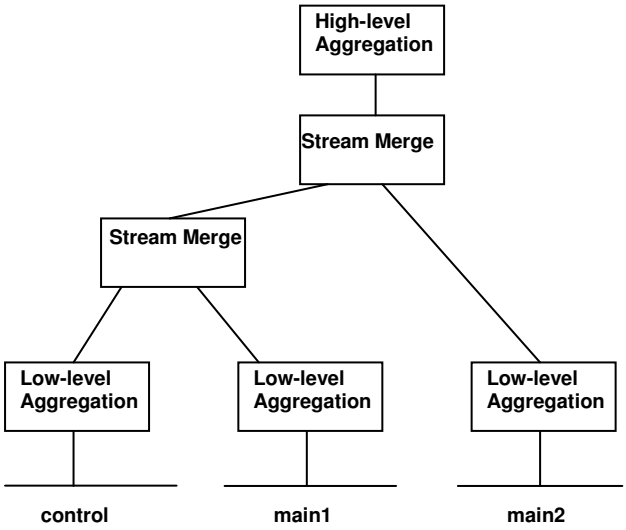

Figure 2: Merge query execution plan

Data is partially aggregated using low-level aggregation queries and then combined using stream merge operators before finally being aggregated by high-level aggregation query. When the *control* link has no traffic, both stream merge operators must buffer a large number of tuples received from high-rate main links. In this experiment, we varied the interval with which heartbeats are generated and recorded maximum memory that a running query consumes. We varied a heartbeat interval from 1 sec (the default value used in Gigascope) to 30 seconds in 5 second increments. The results of the experiments are presented in Figure 4.

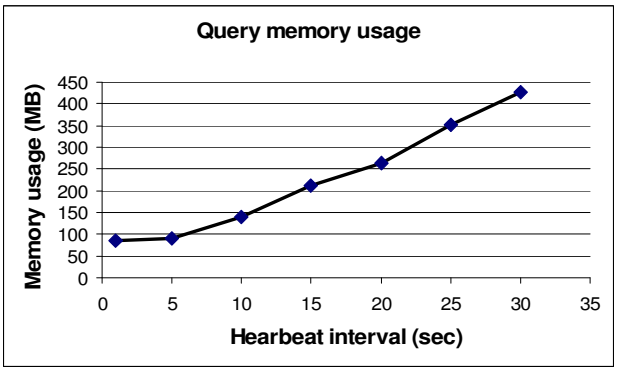

Figure 3: Memory usage of stream merge query

The result of the experiment illustrate that heartbeats successfully unblock the stream merge operators. As the heartbeat interval increases, the amount of state that the merge operators need to maintain before they can advance the epoch is growing linearly. Eventually memory footprint of the query would exceed the available RAM and will cause a system crash.

It is important to notice that increasing the heartbeat intervals not only leads to increased memory footprint, but also significantly increases the amount of data that needs to be flushed by the operator once the epoch advances. Since our stream merge implementation does not currently use traffic-shaping techniques (such as slow flush), the system can cause a query failure even before the memory consumption exceeds the available RAM. In the experiment in which we used 30 second heartbeat intervals, merge operators were instantly flushing 420MB worth of tuples which exceeded the capabilities of tuple transfer mechanism and led to query failure.

7.2 Unblocking join operators using hearbeats

In this experiment we observed how effectively heartbeats unblock join queries and reduce overall query memory requirements. We used the following GSQL query:

```
Query flow1:
SELECT tb,protocol,srcIP,destIP,
      srcPort,destPort,count(*) as cnt
FROM [main0_and_control].DataProtocol
GROUP BY time/10 as tb,protocol,srcIP,
      destIP, srcPort, destPort;

Query flow2:
SELECT tb,protocol,srcIP,destIP,
      srcPort,destPort,count(*) as cnt
FROM main1.DataProtocol
GROUP BY time/10 as tb,protocol,srcIP,
      destIP, srcPort, destPort;
```

```
Query full_flow:
SELECT flow1.tb,flow1.protocol,
flow1.srcIP, flow1.destIP,
      flow1.srcPort,flow1.destPort,
      flow1.cnt, flow2.cnt
OUTER_JOIN FROM flow1, flow2
WHERE flow1.srcIP=flow2.srcIP and
      flow1.destIP=flow2.destIP and
      flow1.srcPort=flow2.srcPort and
      flow1.destPort=flow2.destPort and
      flow1.protocol=flow2.protocol and
      flow1.tb = flow2.tb
```

Two subqueries (flow1 and flow2) compute the flows aggregated in 10 second timebuckets and observed on interfaces *main1+control* and *main2* respectively. The query results are combined using full outer join to generate a final output. The resulting query plan is shown in Figure 5.

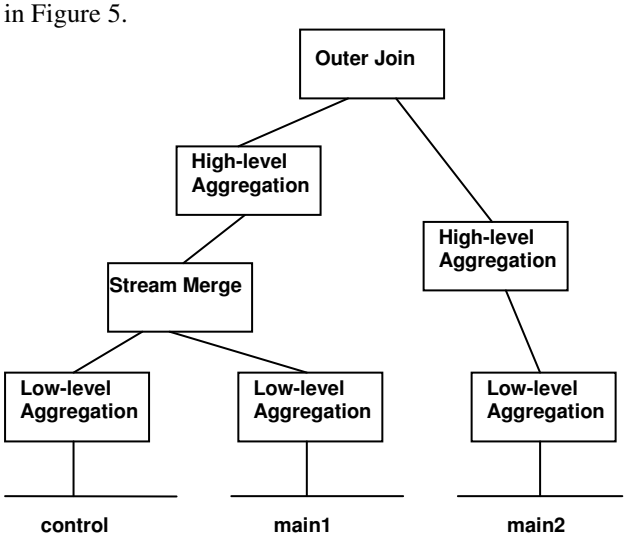

Figure 5: Merge query execution plan

In this experiment we varied an interval with which heartbeats are generated from 1 sec to 60 seconds in 10 second increments. The results of the experiments are presented in Figure 6.

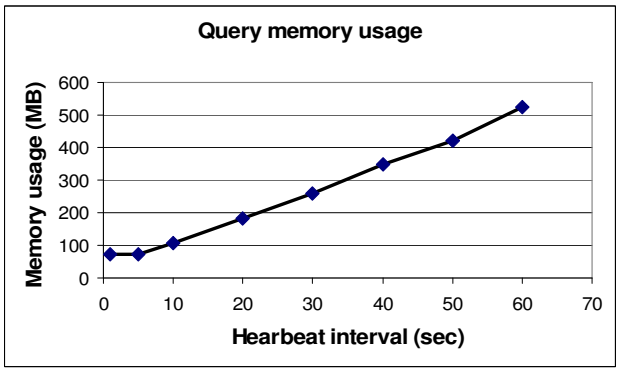

Figure 6: Memory usage of join query

The results of the experiments show a similar pattern to the query that just uses stream merge operators. Again punctuation-carrying heartbeats are able to unblock both merge and join operators. The state maintained by query merge, aggregation and join operators linearly grows with the heartbeat interval and reaches 520MB for 60 second interval. At this point our outer join implementation, which does not use traffic-shaping, instantaneously dumps 520MB of data to receiving application and causes the overflow of system buffers. When we set a heartbeat interval to default value of 1 sec, we not only avoid accumulating large state of blocking operators, but also decreasing the burstyness of their output.

7.3 CPU overhead of heartbeat generaiton

We measured the CPU overhead that Gigascope's implementation of heartbeats incurs on running streaming queries. We measured the average CPU load of a merge query used in Section 7.1 running on two high-rate interfaces (*main1* and *main2*). We compared the CPU load of a system with 1 second heartbeat interval to an identical system which has heartbeats completely disabled. Since both of the monitored links have moderately high load, the merge operators are naturally unblocked even with heartbeat disabled. Therefore both systems behave identically and allow us to measure overhead of heartbeat generation without significantly changing runtime behavior of the operators. We observed that a version of Gigascope with heartbeats disabled has average CPU load of 37.3%, while enabling heartbeat generation every second raises the load to 37.5%. This difference is so small that it can be explained by variations in traffic load. Hence we conclude that the overhead of the heartbeat mechanism is immeasurably small.

8 Conclusion

We introduced a simple mechanism for punctuation-carrying heartbeat generation that we developed for Gigascope, a high-performance streaming database for network monitoring, that is operationally used within AT&T's IP backbone. We show how heartbeats can be regularly generated by low-level nodes in query execution plans and propagated upwards. By attaching temporal update tuples as punctuation, the heartbeats unblock any blocked operators. Our heartbeat mechanism can be also be used for other applications in distributed settings, such as detecting node failures, performance monitoring, and query optimization. A performance evaluation using live data feeds show that our system is capable of working at multiple Gigabit line speeds in industrial deployment and can significantly decrease the query memory utilization.

9 REFERENCES

[1] Daniel J. Abadi et al. The Design of the Borealis Stream Processing Engine, CIDR 2005.

[2] A. Arasu et al. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

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

[4] D. Carney et al. Monitoring streams - a new class of data management applications. In *Proc VLDB*, pages 215–226, 2002.

[5] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. *CIDR* 2003.

[6] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. *CIDR* 2003.

[7] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: high performance network monitoring with an SQL interface. In *Proc. ACM SIGMOD*, page 262, 2002.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD*, pages 647–651, 2003.

[9] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1): pages 27–32, 2003.

[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of the 12th Intl.Conf. on Data Engineering*, pages 152–159, 1996.

[11] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. VLDB*, page 1149, 2003.

[12] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *Data Engineering Bulletin*, 26(1):49–56, 2003.

[13] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. SIGMOD Conference 2005.

[14] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record*, March 2005.

[15] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc.IEEE ICDE Conf.*, 2002.

[16] S. Muthukrishnan. Data streams: Algorithms and applications. In *ACM-SIAM Symp. Discrete Algorithms*, `http://athos.rutgers.edu/muthu/stream-1-1.ps`, 2003.

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

[18] Stanford stream data manager. `http://www-db.stanford.edu/stream/sqr`, 2003. J. Widom and *et al*.

[19] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Technical Conf.*, 1998.

[20] Peter A. Tucker, David Maier, Tim Sheard , Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams, IEEE Transactions on Knowledge and Data Engineering, v.15 n.3, p.555-568, March 2003.

[21] T. Johnson, S. Muthukrishnan, I. Rozenbaum. Sampling Algorithms in a Stream Operator, *SIGMOD* 2005.