Gigascope: A Stream Database for Network Applications

Chuck Cranor, Theodore Johnson, Oliver Spataschek
AT&T Labs – Research
{chuck,johnsont,spatsch}@research.att.com

Vladislav Shkapenyuk
Dept. of Computer Science, CMU
vshkap@cs.cmu.edu

ABSTRACT

We have developed *Gigascope*, a stream database for network applications including traffic analysis, intrusion detection, router configuration analysis, network research, network monitoring, and and performance monitoring and debugging. Gigascope is undergoing installation at many sites within the AT&T network, including at OC48 routers, for detailed monitoring. In this paper we describe our motivation for and constraints in developing Gigascope, the Gigascope architecture and query language, and performance issues. We conclude with a discussion of stream database research problems we have found in our application.

1. INTRODUCTION

Managing a very large data communications network requires constant network monitoring. IP networks are difficult to manage (a side effect of their decentralized nature), protocols and services are becoming more complex (MPLS, VPNs, multimedia, etc.), and network attacks are common. Most network analysis is done via ad-hoc tools on network trace dumps, often resulting in severe data management problems. High speed (gigabit and higher) network monitoring tools are available, but they are inflexible in the types of reports they generate.

Our goal in designing Gigascope was to develop a network data analysis tool which has the speed and flexibility that network analysts require, but which provides a structured querying environment to make complex analysis tractable. We also wanted Gigascope to be adaptible enough that it could be used as the primary data analysis engine in many settings: traffic analysis, performance monitoring and debugging, protocol analysis and development, router configuration (e.g. BGP monitoring), network attack and intrusion detection and monitoring (e.g. distributed denial of service attacks), and various ad-hoc analyses.

To a person in the database community, the need for a structured query environment and adaptibility is self-evident. What might not be so clear is the need for high performance and flexibility. Network analysts tend to be (justifiably) suspicious of using a DBMS for their analysis engine. One significant problem is performance. Even with the high degree of sampling and aggregation in Netflow records (traffic summaries produced by routers) the AT&T IP backbone alone generates 500 Gbytes of data per day (about ten billion fifty byte records). A further problem is that certain portions of the analysis can be very complicated. Many analyses require that a network protocol be simulated, e.g. IP defragmentation or reconstructing TCP sessions. Network analysts have developed special fast algorithms and software to overcome these problems, and are loathe to risk their work on systems which in the past have proven to be slow and inflexible.

Nevertheless, severe data management problems persist. In our interactions with network analysts, we observed that most of their analyses could be expressed as a sequence of SQL queries. Furthermore, data analysis is best done close to the data source to reduce the data volume as soon as possible. Furthermore, if the analysis could be done on-the-fly, we could offer capabilities that current methods cannot provide. We therefore decided that Gigascope [4] should be a lightweight stream query processing system which is at least as fast as a hand-written system and which allows the user to bypass the query system as needed, but which provides sophisticated query and data management facilities.

Stream databases have recently become a popular research topic. We refer the interested reader to the recent survey by Babcock et al. [1], which space constraints do not allow us to repeat here. Much of the recent work on stream databases use a *continuous query* model, evaluated over a *sliding window* (e.g., the query language proposed in [1]). Our query system is different, being a pure stream query processing system more akin to Tribeca [6]. Unlike Tribeca the Gigascope uses an SQL-like language (GSQL) rather than a procedural language to express its queries, allowing query composition and query optimization.

2. QUERY LANGUAGE

The Gigascope query language, *GSQL*, is a pure stream query language with SQL-like syntax (being mostly a restriction of SQL). That is, all inputs to a GSQL are streams, and the output is a data stream. We feel that this choice (akin to that made by Tribeca and Hancock) allows for precise query semantics, enables the composition of comples query processing, and simplifies the implementation of fast operators.

The query model used by most of the recently proposed stream database systems is that of a *continuous query* over a *sliding window* of the data stream. While this model has some advantages (e.g., presentation of results to the end user) and some areas of best application (e.g. sensor networks), we felt that the continuous query model to be inappropriate for network data analysis. One significant problem is that the continuous query model makes query composibility difficult. The input to a query is one or more data streams, but the output is a (continuously changing) table. Queries

SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

can still be composed (i.e., Q_2 can use the output of Q_1 as its input), but the differences in the output of Q_1 must be often reverse interpreted as a data stream.

A second problem is the difficulty of precisely expressing a query – or conversely, understanding what a query means. Let us consider example query Q3 from [1]:

```
(Select Count(*) From C, B
Where C.src=B.src and C.dest=B.dest and C.id=B.id)
/ (Select Count(*) from B)
```

This query is intended to identify fraction of traffic in the backbone B which can be attributed to a customer network C. However the semantics of the result are not clear. Since the output is used for monitoring, the intended result is not likely to be the evaluation of the query over the entire stream, rather over some recent window. However, the window is not specified, and there are in fact three windows to specify (two in the first subquery, one in the second). The snapshots taken by these three subqueries must be precisely synchronized (but on what is not specified), else the result is erratic and meaningless. If the respective windows are defined by a number of tuples rather than by time, the three windows will certainly be unsynchronized.

Although example Q3 appears to be simple, an examination of the evaluation details shows that the semantics are complex. We note that examples Q1 and Q2 in [1] make explicit references to timestamps, while the queries in section 5.1 make explicit or default references to sliding window sizes.

A final reason to use a pure stream query rather than a continuous query model is the simplicity of implementation. Rather than transforming the input data stream into a windowed table, we can operate on the data stream directly.

2.1 Ordered Attributes

One concern in a stream database language is that of *blocking operators*. While some operators (such as selection and projection) need no state other than the tuple being processed, other operators (such as aggregation and join) potentially require their entire inputs before a single output can be produced. One approach to bounding the state is to use sliding windows [1]. Instead, our approach will be to analyze the "timestamps" of the input stream(s) and the properties of the query to determine a query plan which bounds the state required to evaluate blocking operators.

We observed that network analysis data generally contains one or more timestamps or sequence numbers, that these timestamps generally increase (or decrease) with the ordinal position of a tuple in a stream, and that almost all queries make reference to these timestamps. We therefore adopt an approach similar to that of a sequence database. However, a sequence database model has a couple of limitations which make it impractical for our application. First, network data streams often have several timestamps and sequence numbers, and they might not be monotonically increasing with the ordinal position of the tuple in the stream. For a simple example, Netflow records have a *start* and an *end* timestamp. A stream of Netflow records produced by a router will have monotonically increasing *end* timestamps, and generally (but not monotonically) increasing *start* timestamps. Further, most queries on Netflow data will refer to the *start* timestamp rather than the *end* timestamp. The notion of sequence is further perturbed by operators such as join and aggregation. Second, network analysis queries naturally involve predicates and other references to the timestamps and sequence numbers, but not to the ordinal position of a tuple in its stream.

We make use of timestamps and sequence numbers by defining them to be *ordered attributes* having *ordering properties*. These properties might be inherent in the data source, or might be due to processing by an operator. Below is a illustrative but nonexhaustive set of ordering properties:

1. **Strictly/monotonically increasing/decreasing** expresses the usual notion of a timestamp.

2. **Monotone nonrepeating** is a generalization of monotone increasing, and might occur due to a hash function (e.g. Q2 in [1]).

3. **Increasing in group** G : This property states that among the tuples defined by the field in G, the attribute is increasing. This property can occur after aggregation. For example, the *start* time of a Netflow record (an aggregation of packets) is increasing in group $\{$*sourceIP,destIP,sourcePort,destPort,protocol*$\}$.

We might need to modify these definitions to account for almost-sorted input. For example, Netflow records are sorted on the end time, and all Netflow records are dumped every 30 seconds. Therefore the start time of a record is always within 30 seconds of the high water mark, i.e. the *start* attribute is banded-increasing(30 sec.).

We use the ordering attributes to turn blocking operators into stream operators. In the current implementation of Gigascope, we use the monotone increasing property as follows:

- **Join** : The join predicate must contain constraint on an ordered attribute from each table which can be used to define a join window. For example, *B.ts=C.ts* or *B.ts*$\geq$*C.ts-1 and B.ts*$\leq$*C.ts+1.*

- **Group-by and aggregation** : The group key must contain at least one ordered attribute. When a tuple arrives for aggregation whose ordered attribute is larger than that in any current group, we can deduce that all of the current groups are *closed* and will receive no further updates in the future. All of the closed groups are flushed to the output.

The Gigascope data definition language allows the user to specify special properties of the attributes in a source stream, including the ordering properties. The query processing system will impute ordering properties of the output of query operators. For example, suppose that attribute ts is monotonically increasing and a projection operator computes the value $ts/5$ as one of the attributes in its output. We can impute that $ts/5$ is also monotonically increasing. We can perform similar reasoning for the group-by/aggregation operator.

The ordering property imputation for the join operator is more complex, and depends on the constraints in the join predicate and the particular join algorithm selected. For example, if *B.ts* and $C.ts$ are monotonically increasing, *B.ts* is in the output, and the join predicate contains the constraint *B.ts=C.ts*, then *B.ts* in the output will be monotonically increasing. If the constraint is *B.ts*$\geq$*C.ts-1 and B.ts*$\leq$*C.ts+1*, then in the output *B.ts* might be monotonically increasing or banded-increasing(2) depending on the choice of join algorithm (monotonically increasing requires more buffer space).

2.2 The GSQL Language

GSQL is a SQL-like stream database language, being mostly a restriction of SQL but with some stream database extensions. Currently GSQL supports selection, join, aggregation, and *stream merge* (discussed below). Join queries are currently restricted to two-stream joins, and the join predicate must include a constraint which defines a window on ordered attributes from both streams. Aggregation queries *should* have at least one ordered attribute as a one of the group-by keys, but this restriction is not enforced (the user can obtain output by flushing the query).

All queries operate over streams, which come in two flavors: *Protocols*[1] and *Streams*. A Protocol is a data stream generated by interpreting a sequence of data packets which are presented to the Gigascope run time system. These data packets can be from any reasonable source – IP packets transported via OC48, Netflow packets, BGP updates, etc. The Gigascope run time system interprets the data packets as a collection of fields using a library of interpretation functions. The schema of a Protocol stream maps field names to the interpretation functions to invoke. A Stream is the output of Gigascope query. The fields of its tuples are packed in a standard fashion.

A Protocol defines a mechanism for interpreting a data source, but not what serves as the data source (whereas the source of a Stream is the output of a query). To completely specify a data source, the Protocol must be bound to an *Interface* – a symbolic name which the run time system can bind to a source of packets (if no Interface is given, a default Interface is implied). An example which reports the destination IP and port, and a timestamp from TCP packets on eth0 (the first Ethernet interface card) is:

```
DEFINE{ query_name tcpDest0; }
  Select destIP, destPort, time From eth0.TCP
  Where IPVersion = 4 and Protocol = 6
```

The DEFINE section of a query allows the user to set properties of the query. In this case, the query name is set to tcpDest0. A user application or another GSQL query can read the output of tcpDest0 by specifying it in the From clause.

GSQL contains an extension to SQL, the *merge* operator, which is a Union operator which preserves the ordering properties of an attribute. For an example of a merge query, suppose that we have a tcpDest1 which matches tcpDest0 except that it reads from Interface eth1:

```
DEFINE{ query_name tcpDest; }
  Merge tcpDest0.time : tcpDest1.time
  From tcpDest0, tcpDest1
```

The merge operator allows us to combine streams from multiple sources into a single stream. This operator is surprisingly important – we implemented it before the join operator. We developed Gigascope to monitor optical links, which are usually simplex rather than duplex. To obtain a full view of the traffic on a logical link, we need to monitor two interfaces and merge the resulting streams. This query illustrates another feature of GSQL, namely the ease with which queries can be composed into a complex processing chain. GSQL currently supports nested subqueries through this mechanism only, but supporting subqueries in the FROM clause requires only an update of the parser.

GSQL supports the join of two streams as long as it can determine a join window from the join predicates. However, GSQL does not currently support the join of a stream to a non-stream relation. Instead GSQL provides support for user-written functions which can act as special types of (foreign key) joins. These have worked so well in practice that supporting non-stream tables in GSQL has become a low priority.

Users can make new functions available by adding the code for the function to the function library, and registering the function prototype in the function registry. In the function registry, the function can be marked as a *partial* function, meaning that it might not return a value. The processing is the same as if there is no result from a join – the tuple being processed is discarded. One or more of the parameters of the function can be marked as *pass by handle*. These parameters (which must be literals or query parameters) require expensive pre-processing before the function can use them, for example a regular expression to be compiled. Lets consider an example:

```
Select peerid, tb, count(*) FROM tcpDest
Group by time/60 as tb,
  getlpmid(destIP,'peerid.tbl') as peerid
```

The attribute time is a 1-second granularity timer, so *time/60* defines minute-long buckets (when group with a new value of *tb* is produced, all of the pre-existing groups are closed, and therefore are flushed to the output stream). The *getlpmid* function performs *longest prefix matching* – that is, it identifies which subnet an IP address belongs to. Longest prefix matching is a common network analysis activity, and researchers have developed special fast algorithms for it, which getlpmid implements. The second parameter is a pass-by-handle parameter, which indicates a file containing the prefixes of the autonomous systems (AS) of AT&T IP peers (i.e., obtained from a routing table). When the query is first invoked, the parameter handle registration function reads this file and builds a special in-memory for the function (the parameter handle ties this table to the function invocation).

This example illustrates one of the design principles of Gigascope, that it adapt to the needs of network analysts. If we could not support the special algorithms that they have developed, Gigascope would be rejected as slow and inflexible.

3. ARCHITECTURE

The central component of Gigascope is a *stream manager* which tracks the query nodes that can be activated. Query nodes (for example, tcpDest would be a query node) are processes. When they are started, they register themselves with the *registry* of the stream manager. When a user application or query node needs to subscribe the output of a query, it submits the query name to the registry and receives a query handle in return. The process then contacts the query node to set up communication (through shared memory). The stream manager does not track the connection further (which has positive and negative aspects).

The GSQL processor is actually a code generator. A GSQL query is analyzed then translated into either C code or C++ code (the split is discussed below). While a code generation approach results in some loss of flexibility, our experiences with Daytona [5] have shown that it is capable of producing the fastest system.

The generated code interfaces with an API to perform tasks such as registering itself, accessing source streams, and accepting and producing tuples. Users can write their own query nodes to implement special operators by following this API. For example, we have implemented a special IP defragmentation operator in this manner and have built a query tree using it. The ability to bypass the existing query system when necessary is a critical flexibility in our application domain.

Optimizations GSQL optimizes its queries by rearranging the query plan, and by low-level optimizations on the generated code. One significant optimization technique is to push the query as far down the processing stack as possible, even into the network interface card (NIC) itself. This is accomplished in part by breaking queries into high level query nodes (HFTAs) and low level query

[1] This word was chosen because of its connotations to the end-users.

nodes (LFTAs)[2]. All HFTAs accept only Stream input and exist as separate processes, while LFTAs accept only Protocol input and are linked into the stream manager. One reason for this separation is library convenience – the LFTAs must make use of source data packet interpretation libraries which are linked into the stream manager run time system. However, the split is also a performance optimization. The LTAs are lightweight queries which perform preliminary filtering, projection, and aggregation. By linking them into the RTS, these preliminary queries can be evaluated without additional data transfers, and greatly reduce the data traffic to the HFTAs. To an application LFTAs and HFTAs look identical, and in fact a simple query can execute entirely as an LFTA. If the GSQL processor splits a query into an HFTA and an LFTA component, both streams are available to the application, though the LFTA query will have a mangled name.

Depending on the capabilities of the NIC, Gigascope can perform further optimizations. If the NIC has an appropriate RTS, we execute the LFTAs inside the NIC. Other NICS allow us to specify a *bpf* (berkley packet filter) preliminary filter, and to specify the number of bytes of qualifying packets (the "snap length") to be returned (that is, we can push a simple selection/projection operator into the NIC).

An LFTA can perform aggregation, but it uses a small direct-mapped hash table. Hash table collisions result in a tuple computed from the ejected group being written to the output stream. Because of temporal locality, aggregation even with a small hash table is effective in early data reduction. An aggregation LFTA will feed its results to an HFTA, which completes the aggregation. This processing is similar to that of *subaggregates* and *superaggregates* used in data cube computation algorithms. This aggregate query splitting optimization was one of our motivations to build Gigascope as a pure stream database, because its basic optimizations create networks of queries over streams.

Because LFTAs and lower level components are linked into the RTS, and possibly into the NIC, all queries which generate LFTAs must be submitted in a batch. Changing the set of LFTAs requires that the query system be stopped, the RTS changed, and then restarted. However new HFTAS can be submitted at any point. To increase the flexibility of the system queries can accept query parameters, which are similar to constants but which are specified at query instantiation time and which can be changed on-the-fly. The RTS can execute multiple instances of the same LFTA, each with different parameters.

The architecture decision to link the LFTAs into the RTS introduces some inflexibility into Gigascope (mitigated somewhat by query parameters and the fact that we can change the RTS in seconds). However, it produces a signicant benefit. By the lightweight design, the query plan and code generation optimizations, and the earliest possible reduction of data flowing in the system, Gigascope executes as fast as hand-written analysis code (and often much faster).

Unblocking Operators As discussed in Section 2.1, GSQL uses ordering properties of attributes of a stream to make non-blocking implementations of operators such as join, merge, and aggregation – by defining the window on the streams over which the query must execute. However, operators over multiple streams (merge and join) can still block if one of the input streams is slow in providing tuples – perhaps because the slow stream is a naturally low volume one. Consider the merge query in Section 2.2. If tcpDest0 produces 100Mbytes of data per second while tcpDest1 produces one tuple per minute, we are likely to overflow the merge buffers (network traffic is notoriously bursty in this manner).

The problem is that the presence of a tuple allows us to advance the window over which a query operates, but we do not get this information in the absence of a tuple. To overcome this problem, we use a mechanism similar to the one proposed by [7]. of injecting *ordering update* tokens into the query stream. These tokens contain lower bounds on the ordering attributes in the stream. While these tokens are injected periodically by [7], we are experimenting with an on-demand system (i.e., if an operator detects that it might be blocked).

4. PERFORMANCE CONSIDERATIONS

The performance measure for a stream database system is not how fast it can produce answers, but rather how high the data rate can be on its input stream(s) before it starts dropping tuples? This point is well expressed in [2].

To test several performance alternatives we wrote a collection of queries to compute the fraction of port 80 traffic which is due to the HTTP protocol (port 80 is used to tunnel through firewalls). This is done by comparing a count of all packets on port 80 with a count of packets on port 80 whose data payload matches the regular expression `` ``^[^\\n]*HTTP/1.*'' ``. That is, expensive processing is required for the evaluation of this query. Regular expression finding is too expensive for an LFTA, so the filter query was split into an LFTA which filters TCP packets on port 80, and an HFTA part which perform the regular expression matching.

We ran our experiment using a 733 Mhz processor, 2GB of memory, and a Tigon gigabit ethernet card. We generated 60 Mbit/sec of port 80 traffic, and additional background traffic to vary the data rates. We tried four approaches 1) dumping the data to disk for post-facto analysis, 2) reading data from the ethernet card using libpcap, then discarding the packet (best case processing), 3) Running Gigascope with the LFTAs executing in the host (i.e., reading from libpcap), and 4) running Gigascope with the LFTAs executing on the Tigon gigabit ethernet card. We chose a 2% packet drop rate as the maximum acceptable loss.

Option 4) (LFTAs on the NIC) had the best performance, with less than a 2% loss rate even with a 610 Mbit/sec traffic rate (the most that our router could handle). Options 2) and 3) had similar performance, managing 480 Mbit/sec before exceeding a 2% packet loss. At this point the system experienced interrupt livelock. Option 1), dumping the data to disk, had by far the worst performance with a packet loss rate exceeding 2% at only 180 Mbit/sec (dumping data to fast striped disks).

While space constraints prevent us from a detailed discussion of the experiment, several points are clear:

- Early data reduction is critical for performance, and the earlier the better.

- Touching disk kills performance – not because it is slow but because generates long and unpredictable delays throughout the system.

- Contrary to what has been written, an efficient stream database can execute complex queries over very high speed data streams. For example, in [1] the example query Q3 (repeated here in Section 2) is cited as a type of query which requires sampling and approximation. However the query in our experiment requires far more intensive processing, and even the libpcap trial only fails when interrupt livelock occurs. A sufficiently complex query workload will require sampling and approximation, but it is a technique of last resort.

[2] FTA stands for "Filtering, Transformation, and Aggregation". This is another example of avoiding database jargon and adopting instead network analysis language.

In [2], the authors note that not all tuples are equally valuable and that some can be more readily dropped than others. The authors propose a scoring and scheduling mechanism to ensure quality of service. We concur with their position that some tuples are more valuable, but we use a simple heuristic which is easy to understand and implement: highly processed tuples (produced further in the query chain) are more valuable than less-processed tuples, because of the filters and aggregations that have been applied.

5. CONCLUSIONS

We have developed Gigascope [4], a high performance stream database for network applications. Currently we have seven installations, most of which proide special network monitoring services for AT&T customers. Our largest scale deployment monitors application protocol performance over two Gigabit Ethernet links for one of our customers. At peak periods, Gigascope processes 1.2 million packets per second using an inexpensive dual 2.4 Ghz CPU server. At the time of writing, this application has been running for three months nonstop. Additional deployments throughout AT&T, including OC48 deployments, are in negotiations.

By working closely with network analysts, we developed a system which is fast and flexible enough to satisfy their expectations. While SQL is not a familiar language to the analysts, they quickly appreciate the ease with which new monitoring tasks can be implemented. We have found that they soon start writing queries which make aggressive use of language features (the examples in 2.2 are simplified from an user's application) and start demanding more.

Research Directions In the course of developing Gigascope, we have encountered several issues which we have deferred for future work. One issue is that our stream operators provide another dimension in optimization because the choice of operator implementation affects the attribute ordering properties of its output, which in turn affects the performance of downstream operators. Another issue is the proper use of sampling and approximation. Many query sets do not need these techniques, but when they are applied it must be integrated into the query language under the control of the analyst.

While GSQL suffices for a large class of tasks, many network analysis queries find and aggregate subsequences of the data stream (i.e., extract the TCP/IP sessions). We are exploring how to integrate the complex group definition mechanisms described in [3] into GSQL.

The most pressing issue, however, is expressing which streams are the source of the query. Our *Interface.Protocol* mechanism works for our current installations, but will not scale for the planned dozens to hundreds of deployments. Surprisingly, this issue has not received attention in the research literature.

6. REFERENCES

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

[2] D. Carney and et al. Monitoring streams - a new class of data management applications. In *Proc. Intl. Conf. Ver Large Databases*, 2002.

[3] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-join: An operator for complex olap. In *Proc. Inthl. Conf. on Data Engineering*, pages 524–533, 2001.

[4] C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatschek. Gigascope: High performance network monitoring with a SQL interface. Sigmod 2002 demonstration, 2002.

[5] R. Greer. Daytona and the fourth-generation language cymbal. In *Proc. SIGMOD Conf.*, pages 525–526, 1999.

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

[7] P. Tucker and D. Maier. Exploiting punctuation semantics for querying continuous data streams. In *intl. Conf. Data Engineering*, 2002.