Tribeca:
A System for Managing Large Databases of Network Traffic

Mark Sullivan
Juno Online Services
Andrew Heybey
Niksun, Inc.

Tribeca: A System for Managing Large Databases of Network Traffic

Mark Sullivan
Juno Online Services
120 West 45th Street, 15th floor
New York, NY 10036
sullivan@staff.juno.com

Andrew Heybey
Niksun, Inc.
180 Tices Lane
East Brunswick, NJ 08816
ath@niksun.com

Abstract

The engineers who analyze traffic on high bandwidth networks must filter and aggregate either recorded traces of network packets or live traffic from the network itself. These engineers perform operations similar to database queries, but cannot use conventional data managers because of performance concerns and a semantic mismatch between the analysis operations and the operations supported by commercial DBMSs. Traffic analysis does not require fast random access, transactional update, or relational joins. Rather, it needs fast sequential access to a stream of traffic records and the ability to filter, aggregate, define windows, demultiplex, and remultiplex the stream.

Tribeca is an extensible, stream-oriented DBMS designed to support network traffic analysis. It combines ideas from temporal and sequence databases with an implementation optimized for databases stored on high speed ID-1 tapes or arriving in real time from the network. The paper describes Tribeca's query language, executor and optimizer as well as performance measurements of a prototype implementation.

1 Introduction

The rapid growth of high speed computer and telephone networks means that the tools to analyze and engineer the networks are becoming more and more important. Network engineers use a combination of hardware and software tools to monitor the network, record various statistics and flows, and analyze the collected data. These tools either operate directly on the live network or record traffic for later offline analysis. For example, one group we have worked with records OC3 links (155 Mb/s) and groups of 16 T1 links (1.5Mb/s each, 24Mb/s aggregate). Their tape technology ranges from 8mm tape to 96 GByte ID-1 tapes that transfer data at about 256 Mb/s. The data from a monitoring run ranges from a few gigabytes to hundreds of gigabytes. Network engineers expect this number to grow rapidly into the terabyte range as monitoring tools, networks, and storage technologies improve in price and performance.

Network traffic engineers use their vast collections of network data to perform such diverse tasks as protocol performance analysis, conformance testing, error monitoring and fraud detection. In general, each group writes its own ad-hoc programs to examine and analyze the data. Although these programs query large databases of recordings, the traffic engineers avoid using conventional relational database management systems (RDBMSs) for several reasons:

- Both the data and the storage medium are stream-oriented. Fast sequential access to data is crucial; transactional updates, fast access to random records, and concurrency control are not. A highly-tuned C program can outperform a general purpose RDBMS on this workload.
- RDBMSs do not usually handle data on tape well [3]. Non-clustered indices will not work for traffic data. Worse, traffic analysis data is often used only a few times (or once), so load time is a significant cost. Finally, network traffic traces contain many small records with fields a few bits wide, so per-tuple overheads can noticeably increase the database size.
- A network traffic trace is a sequence of time-stamped network protocol headers. The ana-

This work was done while both researchers were employees of Bell Communications Research.

lysts use operators like those found in sequence and temporal DBMSs [13][16]. Traffic analysts usually calculate aggregates on packet interarrival times or calculate and compare network utilization over successive time periods or time scales. The traffic applications also use several data-flow operators and pattern matching operators (e.g. demultiplexing and protocol recognition) that are not common in sequence databases.

- Traffic analysts run batches of related queries during a single pass over the data. Users will sometimes intentionally write queries that use partial results generated by a concurrently executing query. The shared single data source means that even otherwise distinct queries will often share subqueries.
- Users must consider the capacity of the analysis hardware. Often the users would rather reformulate an expensive query or drop an expensive query from the mix than overallocate processor or memory resources in the analysis machine. Relational systems run queries as fast as they can but typically do not provide this kind of capacity information.

Tribeca is a software system for monitoring and analyzing either a live network or recorded network traffic on tape. Tribeca users can write queries to process arbitrarily long streams of information. Like a relational DBMS, Tribeca has a query language that can be compiled and optimized. Like extensible DBMSs [15][5][2], Tribeca has a type system and user-defined operators so it can integrate support for different network protocols and specialized traffic analysis operators. Unlike conventional systems, Tribeca does not support random access to data, transactional updates, conventional indices, or traditional joins.

Tribeca is designed to read a stream of data from a single source (tape or a network interface) and apply compiled queries to the stream. It has a data-flow-oriented query language that allows users to construct large batch queries for the one pass over the data. It also has operations to separate and recombine substreams derived from the source. Finally, Tribeca supports window operators that allow users to compute moving aggregates and to do a very restricted form of join. Both the query language and the optimizer help prevent users from expressing queries that produce intermediate results that cannot be stored in main memory. Because of this, query optimization focuses on memory management and predicate ordering rather than traditional I/O optimizations like access path selection and join optimizations.

Several different groups of network analysts used Tribeca over a one-year period and the system performed well. Measurements show that it is only 1–9% slower than a hand-tuned ad-hoc program on simple queries. With Tribeca, our users are also able to construct more complex queries than they would be able to implement easily in their ad-hoc programs. More importantly, they can easily retarget their queries to do similar analysis on different kinds of networks.

This paper describes the Tribeca design and implementation. Section two gives an overview of the query language. Section three outlines the system's implementation and presents performance measurements from our prototype. Section four compares Tribeca to related work and section five gives conclusions.

2 The Tribeca Query Language

This section describes the syntax and semantics of the query language primitives. Implementation issues are deferred as much as possible until the next section. Before introducing the primitives, we present a motivating example.

2.1 Overview and Example

Figure 1 graphically shows a query used in characterizing IP-over-ATM traffic [7]. The analysts in this case look for “burstiness” in the packet arrival rates and changes in the distribution of packet lengths in order to help plan network capacity. They compare the characteristics of interest over several different time scales (ms, sec, min, hour). In order to isolate a bursty host, they group packet streams by source and destination, calculating similar aggregates over these groups.

The data set for the example is a traffic trace including Asynchronous Transfer Mode (ATM) cells from a dozen virtual circuits (VCs) multiplexed onto the monitored network link. Each trace record contains a time stamp and an ATM cell. The query takes a stream of ATM cells, discards those cells belonging to an uninteresting VC, demultiplexes the stream by

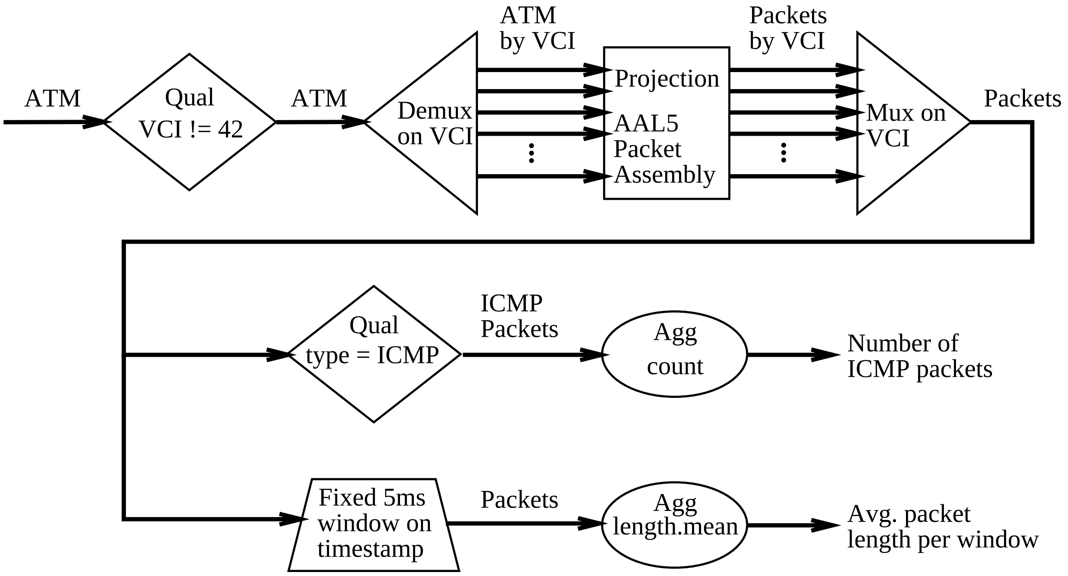

Figure 1: Graphical Representation of a Traffic Analysis Query

virtual circuit identifier (VCI), assembles IP packets (which are fragmented across successive cells on each VC), remultiplexes the packets back into a single stream, then counts the number of ICMP packets and finds the average length of all IP packets in each 5 ms interval.

The example shows several features of the query language presented in this section. Different layers of protocol are implemented by different data types. The example uses simple filters, aggregates, demultiplexing, multiplexing and some simple windows. Each of these is described in subsections below.

2.2 The Tribeca Type System

Tribeca has both a data description language (DDL) and an extensible type system. Like other extensible database managers, the core data management software is type-independent. Data types and operators may be added to the system to support new applications. The procedure for creating extensions in Tribeca is similar to that of Postgres [15] so the details are omitted here. An extension type declaration defines operators and data representations (ascii, host byte order, network byte order) associated with the type. Tribeca allows operator overloading so the same operator name can be used in different data types.

The DDL allows users to create composite types from the compiled-in extension types. The DDL has a simple inheritance mechanism that allows users to describe the kinds of layered packet headers that are commonly found in network traffic data (for instance, UDP/IP and TCP/IP types both inherit from the IP type). In addition to inheritance, the DDL has built-in support for arbitrary offset and width bit fields since network protocols often include bit fields. The DDL has an enumerated type provision so that queries can refer to ID fields by name instead of number (e.g. the field that determines that a routed frame relay packet is transporting IP data has the value 0xCC, but this value can be referred to in a Tribeca query as “IP”). Note that adhoc, unnamed composite types can also be created in queries.

For traffic analysis, support of both an extensible type system and a DDL are crucial. Extensions are needed because some hardware-generated time stamp fields and some fields of network protocols are difficult to describe in a data description language (for example, the DLCI field of a frame relay packet takes several bits from two different bytes of the packet header and combines them into a short int). Extensions are also used to incorporate into Tribeca the exotic statistical estimators used by the traffic analysts. The analysts also want control over the implementation of less exotic estimators, like mean, to ensure that the operator will be numerically stable for their workload. The DDL is important because it allows our non-programmer users to retarget their queries at new networks or at higher levels of the protocol stack without implementing extensions.

The inheritance mechanism and ad-hoc types also help Tribeca queries handle the diversity of higher level protocols used in networks. While at the lowest level all packets on the same network use the same protocol (i.e. a frame relay network carries only frame relay packets), higher level protocols can be quite diverse and their packets interleaved in complex ways. A Tribeca query examines each packet to find out what higher level protocol it uses and then coerces the packet to the appropriate child type and extracts fields of interest from the higher-level part of the packet. This extraction creates an ad-hoc type that can be examined by later parts of the query. We'll see an example of this in the section on multiplexing.

2.3 Streams, Basic Operators, and Simple Queries

Every Tribeca query has a single *source* stream and at least one *result* stream. Stream declarations associate a name with the external source (result) of the stream data, such as a disk or tape file. The *source_stream* statement must also declare the stream's data type. The data types of result streams are derived from the operator that writes to them.

Tribeca supports three kinds of simple operators: qualifications, projections and aggregates. A Tribeca query can combine operators using pipes and transform the stream in several stages. Note that while Tribeca borrows the Unix term "pipe", these are not Unix pipes. Tribeca operators are *not* implemented as separate processes. As the example below illustrates, a Tribeca pipe statement names a stream of data and allows users to express data flow from one Tribeca operator to another.

Qualification operators filter data in a stream. Tribeca's qualification statement specifies a source stream, a result stream, and a list of qualification operators to be applied to the source. Records from the source stream that pass the qualifications are placed on the result stream. While the list of operators is implicitly a conjunction, Tribeca supports the usual complex qualifications involving AND, OR and NOT.

A projection selects one or more fields from each record in the source stream, assembles the fields into a new record, and puts the record onto the result stream. The projection statement may also apply a function to the field during the projection operation. It is important to realize that the projection statement is provided to allow users to construct simple, readable queries. As explained in Section 3, the stream data model allows most projections to be eliminated during compilation. Intermediate tuples are never materialized in Tribeca unless they are used as hash keys or written to external storage.

An aggregate operator is applied to all of the values in a stream and produces a single value. While aggregates of basic Tribeca streams are sometimes useful, queries usually produce streams of related aggregates using demultiplex and window operations described below.

The simple query below uses all operators introduced so far:

```
source_stream s1 is {tape sample1 AtmTrace}
result_stream r1 is {file res1}
result_stream r2 is {file res2}
stream_pipe p1 p2
stream_proj {{s1.atm.ts s1.atm.vci}} p1
stream_qual {{p1.ts.lte 1000000}} r1
stream_qual {{p1.vci.gt 5} {p1.vci.lt 50}} p2
stream_agg {p2.ts.min} r2
```

The query reads a source stream of type AtmTrace from tape. It uses project to create a stream of (time stamp, VCI) pairs (an ad-hoc composite type). That stream is then passed to two different quals. The first saves into a file all (ts,VCI) pairs with timestamp less than 1 million. The second finds all (ts,VCI) pairs in which the VCI is between 5 and 50. Finally, the aggregate finds the minimum time stamp from those pairs.

As described so far, Tribeca queries are trees of stream operations. The source stream may feed any number of operators. Each operator writes to a pipe or a result stream. The source and intermediate streams may feed any number of operators. An intermediate or result stream derives data type from the operator that writes it. None of the simple stream operations introduced so far take their input from more than one stream, but we will introduce operators for combining streams in the next few sections.

2.4 Demultiplexing and Remultiplexing Streams

Traffic analysis queries often must partition a stream into substreams, process the substreams, then recombine the results of the substream analysis. The demultiplexing operation partitions records in a stream based on data in each record. The query below partitions *s1* into substreams of ATM trace records based on virtual circuits, then finds the max time stamp for each VCI:

stream_demux {s1.atm.vci} p1
stream_agg {p1.ts.max} p2

P1 is not a single stream but a set of substreams, each with the cells for one virtual circuit. The pipe *p2* represents a collection of logical substreams each containing the max time stamp for one VCI.

In the example, demux is used like a groupby operator in a relational DBMS. However, the demultiplex operator allows users to apply a series of operations to the demultiplexed streams instead of simply applying aggregates. We can, for example, demultiplex ATM cells by VCI, assemble IP packets from consecutive cell payloads, then apply an aggregate to the IP stream.

In the example below, the query first divides the stream into virtual circuits (*p1*). Each logical substream on *p1* is a sequence of ATM cells for a distinct VCI. Next, consecutive cell payloads from the same circuit are assembled into a stream of IP packets associated with that circuit (usually there are several cells per IP packet. Assembly is actually more complex than this and is described in detail in the subsection on Windows). Finally, the IP stream is qualified and an aggregate is applied to each qualified TCP/IP substream.

stream_demux {s1.atm.vci} p1
stream_proj {{p1.assemble_ip}} p2
stream_qual {{p2.ip_type.eq TCP}} p3
stream_agg {{p3.atm.vci p3.count}} p4

P4 is a set of logical substreams each containing the count of TCP packets for one virtual circuit.

Tribeca allows users to demultiplex the same stream more than once. A second demux simply divides the original stream into more substreams. For instance, we can demux once by VCI then, after assembling IP packets, demux again by IP type (UDP, TCP, etc.). These operations produce substreams that are distinct for each VCI/ip_type pair

In partitioning the stream, the demux operator also "names" each substream. The substream name is not part of the data stream, but may be referred to in project and aggregate operations. In the example above, *p3* is a set of substreams containing <*VCI*, *count*> pairs.

Unfortunately, the demultiplex operator allows users to express queries that cannot be executed in available memory. The demux implementation uses memory space proportional to the cardinality of the demux field. In practice, however, the number of distinct VCIs, packet types and so on in our data is small. Instead of removing demux from the language, Tribeca uses statistics and capacity planning support to help users avoid it when inappropriate. For traffic analysis, demux only really breaks down when users want to partition the packet stream into substreams by time stamp. Streams are long and are typically sorted by time stamp. Users often want to apply aggregates to packets grouped by time value. Tribeca's window feature (described below) allows users to group records by sort field in an efficient way.

2.4.1 Using Multiplex to Combine Streams

The multiplexing primitive is a simple operator for efficiently combining Tribeca streams. It can be used in two ways. In the first usage, mux is used to combine the logical substreams produced by demux. In the second usage, mux combines unrelated streams of the same data type.

Below is an example of the first usage of the multiplex operator. The query uses demux to divide the stream by VCI and counts the packets on each VC. The ⟨ VCI, count ⟩ pairs are then combined into a single stream that is qualified and aggregated.

stream_demux {s1.atm.vci} p1
stream_agg {p1.atm.vci p1.count} p2
stream_mux p2 p3
stream_qual {p3.count.lt 100} p4
stream_agg {p4.count.mean} r1

Note that the mean applied after the mux operates on *all* ⟨ VCI, count ⟩ pairs with count greater than one hundred. Without the mux, the aggregate would have been applied to each virtual circuit separately.

If the stream is demultiplexed more than once, an optional argument to stream_mux allows the substreams to be partially recombined. Suppose a query demultiplexed an ATM cell stream by VCI, assembled it into IP packets (an aggregate), then demultiplexed it again by ip_type. The resulting stream is logically separated by both VCI and ip_type. Muxing by VCI, would leave a stream that was logically divided by only ip_type.

The second kind of multiplex operation, in which several streams are combined, is very common in traffic analysis. Often, several different combinations of the same high level and low level protocols are used in the same network. For instance, frame relay networks have many ways of transporting IP packets (routed, ethernet bridge, fddi bridge, etc.). The query below is a (much simplified) typical stage of frame relay analysis. It finds two types of IP packets, extracts the same interesting fields from both and then combines them into a single stream.

```
stream_qual {s1.is_routed_ip} p1
stream_proj {p1.ts p1.ip_type p1.ip_len} p2
stream_qual {s1.is_bridged_ip} p3
stream_proj {p3.ts p3.ip_type p3.ip_len} p4
stream_mux {p2 p4} p5
```

The output stream is a triple ⟨ time stamp, ip type, length ⟩.

2.5 Windows on Streams

The Tribeca window operator groups successive records in a stream so they may be operated on as a unit. Tribeca supports two kinds of windows. A *fixed* window is effectively a demultiplex operation for the stream's sort field. (Network traffic traces are sorted by time, so the sort field is usually a time stamp). It partitions the stream into non-overlapping groups of records. A *moving* window breaks the stream into successive overlapping groups of records. Each one is illustrated in the example below. Result *r1* contains the number and mean length of large (>100 byte) packets in successive five ms intervals. *R2* contains the inter-arrival time between successive packets:

```
stream_window w1 on s1
  defined by {s1.ts.interval 0.005} is fixed
stream_qual {w1.length.gt 100} p1
stream_agg {p1.count p1.length.mean } r1
stream_window w2 on s1
  defined by {s1.count 2} is moving
stream_agg {w2.ts.diff} r2
```

As the example shows, a Tribeca user-defined function delineates each window. The function may be applied either to the sort field value (time interval) or to the record's position in the stream (count). The window names, *w1* and *w2*, can be used as input to other Tribeca operators in the same way as Tribeca pipes.

In traffic analysis, aggregates are almost always used in conjunction with windows. After every windowfull of data, downstream aggregates produce values and are reinitialized for the next window. Also, packet assembly in ATM analysis is actually implemented using a combination of a window and an aggregate function. The window function is a predicate that returns TRUE on the ATM cell that contains the last byte of the IP packet. The aggregate is a function that combines ATM cell payloads into a single IP packet.

2.5.1 Window Filters

Tribeca streams are like RDBMS relations with one important restriction: Tribeca streams may not be joined. Arbitrary comparisons between records in streams are not allowed for performance reasons. A join operator would permit users to express queries that could be executed only by sorting or repeatedly passing over the data on tape.

The Tribeca *window filter*, however, is a restricted form of join operation that relates records in a window to records in a stream. At any given moment during the execution of a query, the records in a window can be compared to the current record in any stream. For example, the query below searches for cells whose VCI has appeared in a recent predecessor cell. It defines a moving window that records the last 100 atm cells seen, then compares the VCI of the current cell to those of the cells in the window.

```
stream_proj {s1.atm.vci} p1
stream_window w1 on p1
  defined by {s1.count 100} is moving
window_filter {{s1.atm.vci.eq w1}}
  {s1.vci s1.ts w1.ts} r1
```

The window filter arguments are a list of join predicates and a list of projected fields.

While users cannot compare arbitrary records in streams, window filters allow users to compare records that are temporally near one another. Window filters are not as powerful as true joins, but are useful in traffic analysis and can be executed efficiently as long as the window is small enough to fit in memory. For example, one traffic analysis query takes the mean and standard deviation of packet length over N seconds, then uses a window filter to search for packets that are significantly larger than the mean over the next N seconds. We also allow users to initialize windows from files so they can do things like select UDP packets destined for port numbers listed in a file.

3 Implementation and Optimization Issues

While Tribeca's query language and basic data path differ from a conventional DBMS, much of Tribeca's basic software architecture is very conventional. At run time, Tribeca compiles queries into a directed acyclic graph. Each node in the graph contains a list of predicates (like a RDBMS where clause) and a list of project/aggregate operations (like RDBMS target lists). A pipeline is a pointer connecting two nodes. The optimizer rewrites this graph to improve performance. The executor uses it to guide query execution. In the subsections below, we describe some of the Tribeca implementation and optimization strategies.

3.1 Basic Data Management

The stream data model allows Tribeca to perform its basic I/O operations efficiently. I/O in Tribeca consists only of reading the source stream and writing result streams. By design, joins do not affect I/O performance because one join operand must fit in memory. The large sequential read is the dominant I/O cost.

This simple I/O pattern means that Tribeca can take advantage of standard operating system services in a way that a conventional RDBMS cannot. Modern Unix file systems are tuned to detect when applications are doing large sequential reads and support them efficiently. Because a conventional RDBMS accesses data differently from a standard application program, the conventional DBMS must work around operating system resource managers such as the file system and buffer pool. Also, because Tribeca queries do not update data in place, Tribeca needs none of the support and overhead required for concurrency control.

Because of its mix of workloads, a conventional RDBMS faces page size tradeoffs that Tribeca does not; Tribeca has no fixed page size. At run time, it divides available I/O buffer space among its source and destination streams according to the expected volume on those streams (i.e. the source stream is usually very large and the others are small). All streams are double-buffered so that one buffer can be processed while another is being read or written. All read and write operations are in units of full buffers.

Unlike RDBMS's and many other programs, Tribeca cannot preformat its data. That means that it must separate the source stream into records as the data is processed (relational systems use a page structure so repeated scans of the same data do not require record parsing). Tribeca supports record parsing strategies for three different kinds of data: fixed-size records, variable-size records, and "framed" records. The stream data type tells the Tribeca executor which strategy to use. "Framed" streams come from some of the high speed data recorders. Valid records in these streams contain framing patterns that are used to distinguish valid records from periodic bursts of "noise" bytes.

The last important data management issue is that Tribeca makes every effort to minimize internal data copying. Intermediate records produced by Tribeca queries are never unnecessarily materialized. Instead, a pointer to the original field value in the input buffer is used every place the projected value appears as an argument. Materialization occurs only if (a) data is copied to an output buffer (b) data must be aligned correctly for some operator (c) non-contiguous values have to be assembled for a hash key. Also, note that projections involving user-defined functions, cannot normally be eliminated by compilation.

Multiplex and projection operators can often be removed at compilation, sometimes with the help of an extra level of indirection. For example, in frame relay, IP packet headers can appear at different offsets within the packet (depending on whether the frame relay packet is routed or bridged, for instance). Operators downstream from the multiplex must use extra indirection, but the operator does not involve copying.

3.2 Using Coarse-Grain Indices

While Tribeca is largely designed so that a stream of data coming from the network and a stream of data coming from tape can be manipulated in the same way, there is one important difference between the two cases: secondary indices can be constructed for recorded data. Traffic analysis users often identify a few hours of packets that are especially interesting and issue repeated queries over this data. Secondary indices are very important for this workload.

When the database is stored on large, high-speed tapes, however, only a limited kind of index is feasible. Our ID-1 tapes are simply not effective random access devices. That means that we cannot support indices on non-sort fields. Even on the sort field, a full index of every packet in the stream is impractical. The index itself cannot be stored on tape, so it must be much smaller than the data.

Therefore, Tribeca supports what we call *coarse-grained secondary indices*. A coarse-grained index is an approximate index on the sort field of recorded data streams. Users specify the ratio, R, of index size to underlying data size. When building an index, Tribeca inserts a key pointing to one record for every R bytes. The index is always stored on disk.

After the index is constructed, a subsequent query with a qualification on the indexed sort field can use the index to skip over parts of the input data. Because the index is approximate, the scan cursor must be placed on the last indexed record before the scan key. After that, Tribeca applies the qualification to each record until the matching one is found. The search procedure is not as fast as a full index, but it represents a good compromise for the traffic analysis environment. Users can trade off search performance and the size of the index (larger R means smaller indices but a coarser grain search).

3.3 Implementing Fixed and Moving Windows

Tribeca implements fixed and moving windows in different ways. Often, fixed windows require no buffering. If the fixed window is not used in a filter, records are processed through the downstream executor nodes as soon as they arrive in the window. When the fixed window is flushed, Tribeca walks the executor nodes below the window generating aggregate results and reinitializing aggregate nodes for the next window.

A moving window is implemented as a circular buffer. If several windows overlap, the largest window determines the size of the circular buffer. The other windows are represented by pointers into the buffer required by the larger window. Like SEQ [14], Tribeca allows extension implementors to define *moving aggregate functions*. A normal aggregate function takes a single record as an argument (the “current” record in the stream). A window flush causes the normal aggregate function to be applied once for every record in the window. A moving aggregate function is called once per window flush and takes as arguments pointers to the records entering and leaving the window.

Tribeca includes two implementations of window filter: one based on hashing and the other on nested loop. In the first case, the window contents are used to build a hash table and the stream argument probes the hash table. In the second, Tribeca iterates over the window contents for each element in the stream. The hash table is more effective if the window is large and relatively stable. For small windows, the cost of creating and destroying the hash table overrides the cost of iterating over the window contents.

3.4 Optimization Issues

Query optimization in Tribeca has two competing goals. The first is to minimize query execution time. The second is to ensure that the intermediate state associated with the query fits in main memory. Tribeca approaches the first goal like a standard RDBMS. For the second goal, it must pay special attention to two Tribeca operators because they require a data-dependent amount of memory. First, a demux operator is implemented as a hash table whose size depends on the cardinality of the demultiplex field. Second, moving windows and window filters require buffer space to hold their contents. The buffer is data-dependent if the window size is a function of the sort field.

In both space- and time-based optimization, Tribeca benefits from some standard RDBMS optimizations. As is often the case in an RDBMS, pushing qualifications towards the source of the data is helpful so Tribeca migrates quals towards the source stream. Successive qualifications are then grouped together so that the optimizer can use constraint minimization to eliminate redundant quals and order the predicates to minimize expected cost. Migration is, of course, not always possible. For instance, a qual cannot move past an aggregate operator and cannot be moved past a positional window. Tribeca also does some simple common sub-expression elimination. If two instances of the same operator have the same input, they are combined. It should eventually be possible to combine and eliminate larger sequences of operators.

Qual migration can help reduce the number of tuples in windows and reduce the cardinality of demuxes. To further minimize the storage cost of windows that require buffering, Tribeca does several things. First, it combines overlapping windows when it can. Second, it migrates any functional projections whose inputs are smaller than their outputs in front of the window. Still, overflow can occur. On overflow, Tribeca drops records entering the full buffer rather than allow disk I/O. Better user control of overflow error handling is an area for future work.

3.5 Performance Measurements

We measured Tribeca on a Sun Sparc 10 performing queries on two datasets. One data set consisted of frame relay traffic (carrying mostly IP), and the other was classical IP-over-ATM [7] traffic. We used data stored on disk (using the standard SunOS UFS filesystem) and ID-1 tape to perform our measurements. The measurements were run on 260 megabyte disk files and a 10 gigabyte tape file. All tests were run in single-user mode to prevent other user activity from affecting the test results.

We ran a variety of queries of increasing complexity to measure Tribeca's performance as its tasks became more compute-intensive. The queries exercised most of Tribeca's features. Table 1 describes the queries run and the measurement results for Tribeca running on 260 megabyte disk files. In the table the results are given as kilobytes per second and records per second. For ATM the records are 64 bytes long while for frame relay the records are variable length. As a baseline, we also present the speed at which the Unix `dd` program can read the disk file. The `dd` program simply reads the file in blocks whose size is specified by the user (we used the same block size for `dd`, the stand-alone programs and Tribeca), and then writes to the Unix null device (`/dev/null`).

Tribeca compares favorably to the baseline for all of the frame relay measurements, ranging from 1.3 to 5.8 percent slower. The speed decrease is because Tribeca must perform some processing on each record in the file while `dd` does not touch the data at all. When processing ATM cells, Tribeca is considerably slower than the baseline. The records in the ATM data set are much smaller than the frame relay records and the ATM cell record format includes several bit fields that must be reconstructed to perform the query. Tribeca thus spends correspondingly more time processing each record. In these tests Tribeca used 70–75% of the workstation's CPU while `dd` used about 68%.

We estimated the overhead introduced by Tribeca by comparing its performance on several sample queries (`frame1` through `frame4`) to that of a simple stand-alone C program performing the same functionality. The stand-alone program is hard-coded to perform only the tested query so it does not have any of the overhead required to execute a general-purpose query. Table 2 gives the comparison between Tribeca and the stand-alone programs. The table shows that Tribeca introduces no more than 5% overhead relative to the stand-alone program in the test queries. The stand-alone program used about 5% less CPU time than Tribeca.

Finally, Table 3 compares the performance of Tribeca and a stand-alone program when running from a relatively large (10 gigabyte) dataset on ID-1 tape. We only compared Tribeca to a stand-alone program on two queries because of the time required to run the tests. In this case Tribeca ran between 5 and 9 percent slower than the stand-alone program. Both Tribeca and the stand-alone programs used about 98% of the CPU in these tests. The percent CPU used is much higher in the tape tests because the HiPPI interface used to connect to the ID-1 tape drive uses programmed I/O. The device driver must copy each word of data coming from the

Query	Description	Records/sec	KBytes/sec
dd	The Unix dd program reading the disk file.	N/A	5754
frame1	Count all the records in the frame relay trace and sum the length fields of each record.	19130	5423
frame2	As frame1 but qualify on the message type of the record (the message type is the high four bits of a byte in the header).	19992	5667
frame3	As frame2 but further qualify on two character fields (the control and nlpid fields of the frame relay header).	19992	5667
frame4	As frame3 but qualify on two more single-byte fields, the T1 interface and channel (the recorder has multiple interface boards, each board has multiple T1 lines and each line can have multiple channels).	20021	5675
frame5	Qualify to select only IP packets, then demux by source and destination port and count and sum lengths.	19968	5660
frame6	Qualify to select only IP, then demux by T1 board, interface and channel. Count the packets and sum the lengths on each group.	20021	5675
frame7	Qualify by message type and protocol to get two streams of IP packets in slightly different formats (frame relay can carry IP in several different formats). Project the IP protocol field and packet length from each stream and multiplex together to count the total number of IP packets and bytes (regardless of the format used).	19764	5602
frame8	As frame7 but include the board and channel in the projection and then demux by board and channel to count IP packets on each channel.	19816	5617
atm1	Demux by VCI and count cells in each.	50137	3133
atm2	Demux by VCI/VPI and count cells in each.	45098	2818
atm5	Qualify to select a particular VCI and count cells.	76746	4796
atm6	Qualify to select a particular VCI/VPI and count cells.	77425	4839

Table 1: Performance Results for Queries run on Disk

tape. For these tests over 95% of the CPU time is system time.

Our measurements demonstrate that Tribeca adds (in the worst case) 9% more processing overhead than a special purpose program tailored to perform the same query. In most of the tested queries, Tribeca performed even better. The small cost is far outweighed by the flexibility and convenience of changing small simple queries rather than re-writing C code to perform different analyses.

4 Related Work

The difficulties in using relational databases stored on tape are overviewed in [3]. Sarawagi [11] modifies a relational query optimizer to consider large tape archives in its cost formula and caches tape data on faster storage. Video-on-demand systems [4] might use tape storage, but in these workloads many users randomly access independent large objects instead of sequences of small ones.

A temporal DBMS usually treats time as an additional dimension [16]. Implementation issues involve multidimensional indices [6] and disk-based temporal joins [8]. Network traces are temporal, but Tribeca treats it as a one-dimensional stream. Also, so far, the traffic analysts have treated data as events rather than intervals, making temporal joins simpler in Tribeca. Illustra [17] implements time-series data as an ADT, but does not have operators like demux required for traffic analysis.

The SEQ sequence DBMS [12][13][14] integrates sequence operations into an RDBMS. It has operators analogous to those defined in Tribeca although the data flow style of Tribeca's query language should make constructing large batches of sequence queries

Query	Tribeca		Stand-alone program		Ratio
	Records/sec	KBytes/sec	Records/sec	KBytes/sec	
frame1	19130	5423	20151	5712	0.95
frame2	19992	5667	20312	5757	0.98
frame3	19992	5667	20256	5742	0.99
frame4	20021	5675	20363	5772	0.98

Table 2: Tribeca Compared to Stand-alone Programs. The queries are as described in Table 1. The "Ratio" column is the ratio of Tribeca's performance to that of the stand-alone program.

Query	Tribeca		Stand-alone program		Ratio
	Records/sec	KBytes/sec	Records/sec	KBytes/sec	
frame1	27638	6427	30296	7045	0.91
frame4	28944	6730	30412	7072	0.95

Table 3: Tribeca Compared to Stand-alone Program on ID-1 Tape. The queries are as described in Table 1. The "Ratio" column is the ratio of Tribeca's performance to that of the stand-alone program.

easier. Because Tribeca eliminates features of the RDBMS that would slow sequence queries, it runs considerably faster than SEQ on sequence data. However, Tribeca will not support queries that mix relational and sequence data as SEQ does. Also, many of the SEQ optimization strategies involve teaching the relational optimizer to distinguish sequences from relations so the executor can access and buffer them differently. Because it does not support relations, Tribeca's optimizer does not face these problems. For example, Tribeca implements a window primitive instead of teaching the optimizer to distinguish a relation joined to itself from a window scan.

The Tangram system [10] implemented in Prolog has operators similar to Tribeca's basic stream processing operators. Tangram did not include more complex operators such as demux, mux, window, and window filter. Tangram was important early work in stream processing. However, processing stream data in a Prolog system has some potential performance drawbacks. The rule processing in Prolog systems is typically made efficient by carefully-tuned main memory data structures; they do not handle data on secondary or tertiary storage well. Recognizing this, the Tangram project used a relational system as a front-end to handle the bulk of the I/O processing and filtering for the prolog backend. Still, implementing both data management and stream processing together in the same engine will reduce data handling overheads. It will also allow users to write queries in a single query language instead of composing them partially in SQL and partially in Prolog. Further, as in SEQ, trying to implement two kinds of systems in one program will inevitably lead to performance compromises.

There have been several efforts at querying live networks. Datacycle [1] used a specialized network interface to query data circulating through a high speed local network. The Berkeley packet filter [9] allows users to load simple filters into the operating system kernel to generate qualified packet traces efficiently.

5 Conclusions

Network traffic analysis is used in a variety of applications ranging from performance analysis and network provisioning to fraud detection. The characteristics of the data sets and the analyses to be performed on them do not lend themselves to the use of a conventional DBMS, but writing custom programs for each query performed on the data is not desirable either. The data sets are extremely large (the IP-over-ATM trace mentioned earlier in the paper consists of approximately 176 GBytes of data) and come pre-ordered by timestamp. The only practical way to cope with the data is to either analyze it in real time as it happens or to record it on tape.

Tribeca is a stream-oriented database management system designed to support network traffic analysis. Its query language has a data flow character that is familiar to network analysts and supports sequence operators they use in their work. Tribeca's executor is tuned for sequential I/O and the optimizer is focused toward memory and processor limitations rather than join ordering and access path selection. The current implementation of Tribeca has performed useful analyses on several large data sets. The overhead introduced by Tribeca relative to a special-purpose analysis program is not very large, i.e. the convenience and flexibility of Tribeca provide more than enough incentive for analysts to use it.

Acknowledgments

We would like to thank Yatin Saraiya, Sid Devadhur, Paul England, Daniel Barbara, Parag Pruthi, Walter Willinger, Bob Sherman, Namon Jackson, Andy Ogielski, Ravi Jain, and Debbie Swayne.

References

[1] T. Bowen et al. The Datacycle Architecture. *Communications of the ACM*, 35(2), December 1992.

[2] M. Carey et al. Object and file management in the exodus extensible database system. In *Proc. 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[3] M. Carey, L. Haas, and M. Livny. Tapes hold data, too: Challenges of tuples on tertiary store. In *Proc. ACM SIGMOD Conference*, 1993.

[4] A. Chervenak, D. Patterson, and R. Katz. Storage systems for movies-on-demand video services. In *IEEE Mass Storage Symposium*, 1995.

[5] L. Haas et al. Starburst midflight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[6] C. Kolovson. Indexing techniques for historical databases. In *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings Publishing Co., 1993.

[7] M. Laubach. Classical IP and ARP over ATM. Request for Comments 1577, Internet Engineering Task Force, January 1994.

[8] C. Leung and R. Muntz. Query processing for temporal databases. In *Proc. IEEE Conference on Data Engineering*, 1990.

[9] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter USENIX Technical Conference*, Jan 1993. San Diego.

[10] D. S. Parker. *Stream Data Analysis in Prolog*, chapter 8. MIT Press, 1990.

[11] S. Sarawagi. Query processing in tertiary memory databases. In *Proc. Conference on Very Large Data Bases*, 1995.

[12] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. ACM SIGMOD Conference*, 1994.

[13] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *Proc. IEEE Conference on Data Engineering*, 1995.

[14] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. 1996 VLDB Conference*, Mumbai, India, September 1996.

[15] M. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[16] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segeve, and R. Snodgrass, editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings Publishing Co., 1993.

[17] Illustra Information Technologies. Illustra timeseries data blade. Information available via HTTP from www.illustra.com.