Using Data Stream Management Systems for Traffic Analysis – A Case Study –

Thomas Plagemann[2, 1], Vera Goebel[2, 1], Andrea Bergamini[1], Giacomo Tolu[1], Guillaume Urvoy-Keller[1], Ernst W. Biersack[1]

[1]Institut Eurecom, Corporate Communications, 2229 Route des Crêtes
BP 193 F-06904 Sophia Antipolis Cedex, France
{bergamin, tolu, urvoy, erbi}@eurecom.fr

[2]University of Oslo, Department of Informatics,
Postbox 1080 Blindern, 0316 Oslo, Norway
{plageman, goebel}@ifi.uio.no

Abstract. Many traffic analysis tasks are solved with tools that are developed in an ad-hoc, incremental, and cumbersome way instead of seeking systematic solutions that are easy to reuse and understand. The huge amount of data that has to be managed and analyzed together with the fact that many different analysis tasks are performed over a small set of different network trace formats, motivates us to study whether *Data Stream Management Systems* (DSMSs) might be useful to develop traffic analysis tools. We have performed an experimental study to analyze the advantages and limitations of using DSMS in practice. We study how simple and complex analysis tasks can be solved with TelegraphCQ, a public domain DSMS, and present a preliminary performance analysis.

1 Introduction and Motivation

The number of tools for analyzing data traffic in the Internet is continuously increasing, because there is an increasing need in many different application domains. For example, network operators and service providers need to monitor data traffic to analyze the provided service level, to identify bottlenecks, and initiate appropriate counter measures, if possible. This is especially important in the Internet, because the amount of data traffic continuously increases and the behavior and requirements of end-users are changing over time, like accepted response time from web servers. Another application domain is the development and improvement of new protocols and applications, like overlay networks and peer-to-peer (P2P) file sharing applications. The complexity of these protocols and applications, as well as the complexity of the environment they are used in, often impose that a meaningful analysis can only be done during their operation in the Internet. The typical coarse grain architecture of these tools consist of two components, first, a packet capturing or flow statistic component like TCPdump or NetFlow, and second, an analysis component to examine the resulting traces and draw certain conclusions.

Performing traffic analysis to gain new knowledge is normally an iterative process. Traffic analysis tools are used to get a better understanding of network dynamics, protocol behavior, etc. Based on these new insights and influenced by changes in the Internet, e.g., traffic mix and end-user behavior, new analysis goals are defined for the next iteration step. For example, in our recent BitTorrent work, we measured the average throughput of *leechers* (i.e., clients that have not completed the download) by analyzing trace files and identified their origin country [4]. We saw that most leechers are either from the US, Canada, Netherlands, or Australia. Therefore, we analyzed in the next step the average throughput for these countries.

It is a common practice for this type of research to either change or extend existing traffic analysis tools, if it can be avoided to develop a new one. Since many tools are implemented as PERL scripts this often means to study a PERL script and change it. This is not an easy task, even for the author of the script itself if it is not well documented, because the variables in PERL scripts do not have meaningful names. In other words, many problems are solved in an ad-hoc, incremental, and cumbersome way instead of seeking systematic solutions that are easy to reuse and easy to understand. Obviously, the more iterations a traffic analysis study comprises the bigger are the advantages of easily reusable tools.

Another problem with today's tools is the huge amount of data that is generated. Trace files should be archived in order to use them at a later point in time for further studies and evaluations. However, the tools themselves do generally not provide any support for managing these large amounts of data. Therefore, trace files are typically archived as plain files in the file system. Depending on the amount of trace files and the discipline of the researcher to annotate trace files, the retrieval of a particular trace file that is a couple of months old can represent a non-trivial problem.

The huge amount of data that has to be managed and analyzed together with the fact that many different analysis tasks are performed over a small set of different network trace formats, motivates us to study whether Database Management Systems (DBMSs) might be a useful platform for developing tools for traffic analysis. DBMSs are designed to separate data and their management from application semantics to facilitate independent application development. They are designed to handle very large amounts of data, and to efficiently perform searches, comparisons, ordering, summarize data, etc. Furthermore, Internet traffic consists of well-structured data, due to the standardized packet structures, and can therefore easily handled with DBMSs.

Traditional DBMS need to store the data before they can handle it. However, many application domains would benefit from on-line analysis and immediate indication of results. Therefore, we focus our attention on a new emerging DBMS technology called *Data Stream Management Systems* (DSMSs). In contrast to traditional DBMSs, DSMSs can execute continuous queries over continuous data streams that enter and leave the system in real-time, i.e., data is only stored in main memory for processing. Such data streams could be sensor readings, stock market data, or network traffic [1].

Since DSMSs are a promising technology for traffic analysis, we have performed an experimental study to analyze the advantages and limitations of using public domain DSMSs in practice and report our experiences in this paper. In the following section, we discuss in more detail our expectations and requirements on using DSMSs for traffic analysis and describe the approach of our study. We give in Section 3 a brief introduction to DSMSs and the particular DSMS TelegraphCQ [5], [9] we are using. In Section 4, our experiments and their results are presented. In Section 5, we conclude with a general discussion of advantages and limitations and an outlook to our ongoing and future research in this area.

2 Expectations, Requirements, and Approach

Our main expectation with respect to the use of DSMSs for traffic analysis is that DSMSs might be a generic platform that simplify the development of analysis components, are easily reusable, are self-documenting, allow on-line and off-line analysis with the same tool, and support management and archival of data. Typical tasks are to analyze:

- the load of a system, e.g., how often are certain ports, like FTP, or HTTP, of a server contacted; which share of bandwidth is used by different applications; which departments use how much bandwidth on the university backbone,
- characteristics of flows, like distribution of life time and size of flows; relation between number of lost packets and life time of flows; what are the reasons for throughput limitations, or
- characteristics of sessions, like how long do clients interact with a web server; which response time do clients accept from servers; how long are P2P clients on-line after they have successfully downloaded a file.

The above examples indicate an important functional requirement, i.e., the system should be capable to handle all protocol layers including the application layer. An important non-functional requirement is introduced by the need for real-time analysis. A DSMS should be able to handle data with a throughput that is proportional to the network load. For example, to analyze IP and TCP headers on a fully utilized gigabit/s network would require to handle more than 42 megabit/s of data (assuming a fixed packet size of 1500 bytes and a header size of 64 bytes). However, a more realistic assumption is that it is only possible to reduce the data stream by a factor of 4:1 to 9:1 relative to the current network load [6].

DSMSs require like any other DBMS a schema describing the type and structure of the data to be handled. Therefore, we expect that reuse and changes of DSMS applications for traffic analysis will be easier than changing PERL scripts. A side effect of this property might also be that applications can be easier exchanged between researchers and results from others can be easier reproduced.

The major publications in the research area of DSMSs raise the expectations that the above requirements can be met by DSMSs. Even quite high performance numbers are reported for a proprietary DSMS from AT&T, called GigaScope [2]. It has been successfully used for network monitoring and is able to handle at peak periods 1.2 million packets per seconds on a dual 2.4 Ghz CPU server.

We are interested in whether public domain DSMS technology is already mature enough to be useful in practice for traffic analysis. Therefore, we used a public-domain DSMS, called TelegraphCQ [5], and studied how simple and complex analysis tasks can be solved with it.[1] Since it is not the goal of this work to develop new solutions for complex tasks, we investigate how the functionality of an existing tool can be re-implemented with TelegraphCQ. We selected the tool T-RAT [10], because we are using and improving it in our ongoing research work [7].

3 Data Stream Management Systems

The fundamental difference between a classical DBMS and a DSMS is the *data stream model*. Instead of processing a query over a persistent set of data that is stored in advance on disk, queries are performed in DSMSs over a data stream. In a data stream, data elements arrive on-line and stay only for a limited time period in memory. Consequently, the DSMS has to handle the data elements before the buffer is overwritten by new incoming data elements. The order in which the data elements arrive cannot be controlled by the system. Once a data element has been processed it cannot be retrieved again without storing it explicitly. The size of data streams is potentially unbounded and can be thought of as an open-ended relation. In DSMSs, *continuous queries* evaluate continuously the arriving data elements. Standard operator types that are supported by most existing DSMSs are filtering, mapping, aggregates, and joins. Since continuous streams may not end, intermediate results of continuous queries are often generated over a predefined *window* and then either stored, updated, or used to generate a new data stream of intermediate results [1]. Window techniques are especially important for aggregation and join queries. Examples for DSMSs include STREAM [1], GigaScope [2], and TelegraphCQ [5]. The interested reader can find an extensive overview on DSMSs in [1] and [3].

TelegraphCQ is characterized by its developers as "a system for continuous dataflow processing" that "aims at handling large streams of continuous queries over high-volume highly variable data streams" [5]. TelegraphCQ is based on the code of the relational DBMS PostgreSQL and required major extensions to it to support continuous queries over data streams, like adaptive query processing operators, shared continuous queries, and data ingress operations. We focus in this paper on the extensions visible for user, i.e., data model and query language extensions. The format of a data stream is defined as any other PostgreSQL table in PostgreSQL's Data Definition Language (DDL) and created with the `CREATE STREAM` command before a continuous query can be launched and processed.

Figure 1 illustrates the data flow during processing of a continuous query and shows also the main components of TelegraphCQ. The Wrapper ClearingHouse (WCH) is responsible for the acquisition of data from external sources, like the packet capturing tool TCPdump. The WCH loads the user-defined wrapper function for the source. The wrapper is reformatting the output of the source into the PostgreSQL data types according to the DDL definition of the particular stream. There is always a one-to-one relation between source and wrapper, and between wrapper and stream. The WCH can manage multiple wrappers respectively streams and fetches the

[1] At the beginning of this project (August 2003), we identified TelegraphCQ as the only available public domain DSMS.

stream data via TCP connections from the wrapper(s). The newly created tuples of each stream are placed by the WCH into the shared memory infrastructure to make them available to the rest of the system. The query processing is performed in the BackEnd and the results are placed in corresponding queues in the shared memory infrastructure. Finally, the FrontEnd continually dequeues the results and sends them to the client. In order to reuse results, it is necessary to start TelegraphCQ in such a way that the standard output is written to a file.

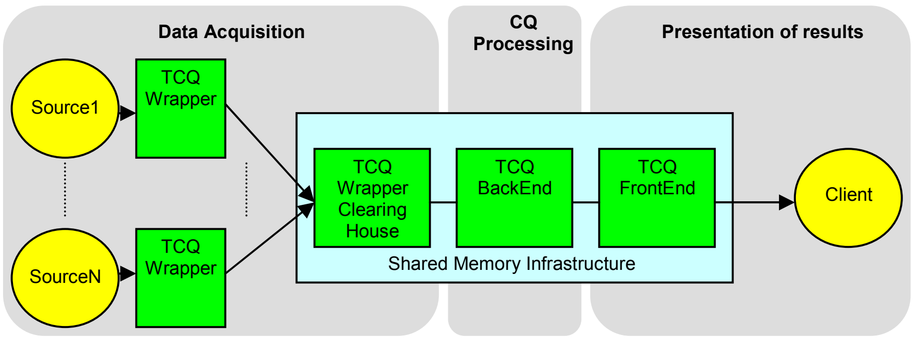

Fig. 1. Data flow in TelegraphCQ (TCQ) during continuous queries (CQ) processing

All interactions between client and TelegraphCQ are performed through the Front-End, including processing of DDL statements and accepting queries from the client. Queries over tables are only directly processed in the FrontEnd. Continuous queries, i.e., queries over streams (and stored tables) are pre-planned by the FrontEnd and passed to the BackEnd.

Continuous queries over data streams are written in SQL with the SELECT statement, but only a subset of the full SQL syntax is supported. The modified SELECT statement has the following form:

```
SELECT <select_list>
FROM <relation_and_pstream_list>
WHERE <predicate>
GROUP BY <group_by_expressions>
WINDOW stream[interval], ...
ORDER BY <order_by_expressions>;
```

Continuous queries may include a `WINDOW` clause to specify the window size for the stream operations. A window is defined in terms of a time interval. Each arriving data element is assigned by the wrapper a special time attribute (called `TIMESTAMP`) and the window borders are continuously updated with respect to the timestamp of the most recently arriving data element, i.e., evaluation of continuous queries is based on *sliding windows* [5]. All other clauses in the `SELECT` statement behave like the PostgreSQL select statement with the following additional restrictions in the TelegraphCQ 0.2 alpha release [9]: windows can only be defined over streams (not for PostgreSQL tables); `WHERE` clause qualifications that join two streams may only involve attributes, not attribute expressions or functions; `WHERE` clause qualifications that filter tuples must be of the form attribute operand constant; `WHERE` clause may only contain AND (not OR); subqueries are not allowed; `GROUP BY` and `ORDER BY` clauses are only allowed in window queries.

4 Experiments and Experiences with TelegraphCQ

In this section, we describe some simple, but typical traffic analysis tasks, the design of a complex analysis tasks, and give some performance bounds for TelegraphCQ.

4.1 Solving Simple Traffic Analysis Tasks

For each of the tasks that are discussed in this subsection, we explain how it can be solved online with a continuous query in TelegraphCQ, or why it cannot be solved with a continuous query. For all examples, we assume an input stream that has been defined with the following DDL statement:

```
CREATE STREAM p6trace.tcp (ip_src cidr, ip_dst cidr, hlen
bigint, tos int, length bigint, id bigint, frag_off bigint,
ttl bigint, prot int, ip_hcsum bigint, port_src bigint,
port_dst bigint, sqn bigint, ack bigint, tcp_hlen bigint,
flags varchar(10), window bigint, tcp_csum bigint, tcqtime
timestamp TIMESTAMPCOLUMN) type ARCHIVED;
```

Each tuple in the stream `p6trace.tcp` comprises IP and TCP header fields and an attribute for timestamp values that are assigned by the wrapper. The attribute type `cidr` defines an attribute to store an IPv4 or IPv6 address in dotted notation.

Task 1. *How many packets have been sent during the last five minutes to certain ports?*
This is an example for a join operation between a stream and a table. The port numbers of interest are stored in the table `services`. The TCP destination port field `port_dst` in all tuples in the stream `p6trace.tcp` is compared with those in the table.

```
CREATE TABLE services (port bigint, counter bigint);

SELECT services.port, count(*)
FROM p6trace.tcp, services
WHERE p6trace.tcp.port_dst=services.port
GROUP BY services.port
WINDOW p6trace.tcp ['5 min'];
```

This is also an example for the use of the sliding window. Each new arriving tuple is advancing the sliding window and the intermediate result of the continuous query for this window is passed to the client. This is a powerful feature for on-line monitoring, since it makes sure that the client receives always the most recent statistic. The drawback of sliding windows is the inherent redundancy in the intermediate results. Assuming that the one minute window covers always *n* tuples, each tuple will contribute *n* times to an intermediate result. This increases the amount of output data and makes it impossible to perform absolute statistics over a stream, like it is necessary for Task 3.

Task 2. *How many bytes have been exchanged on each connection during the last minute?*

```
SELECT ip_src, port_src, ip_dst, port_dst, sum(length-ip_len-
tcphlen)
FROM p6trace.tcp
GROUP BY ip_src, port_src, ip_dst, port_dst
WINDOW p6trace.tcp ['1 min'];
```

From the stream `p6trace.tcp`, all tuples, i.e., packet headers, that have arrived during the last minute are grouped according to their source and destination IP address and their port numbers. The `SELECT` statement specifies that all address information for each group together with the sum of the payload length of all packets in this group are returned. The identification of connections in this continuous query is based on a simple heuristic: during a one minute window all packets with the same sender and receiver IP addresses and port numbers belong to the same connection.

Task 3. *How many bytes are exchanged over the different connections during each week?*

There are two basic deficiencies in the current TelegraphCQ prototype that make it impossible to solve this task with a continuous query. The first deficiency is that a `GROUP BY` clause can only be used together with a `WINDOW` clause. It is obvious that the window size must be much smaller than one week, because all incoming data is kept in main memory until the entire window has been computed. The sliding window imposes that the payload of each packet would contribute several times to intermediate results when the inter arrival time of packets is smaller than the window size. In order to calculate a correct final result that summarizes all intermediate results, it is necessary to remove the redundant information from all the intermediate results. In other words, to calculate absolute statistics with continuous queries, non-overlapping windows (also called *jumping* or *tumbling windows*) are needed, but this type of windows is not supported in TelegraphCQ.

The second deficiency relates to connection identification. The simple heuristic we used for Task 2 to identify connections cannot be used, because the same quadruple of sender and receiver IP addresses and port numbers can identify many different connections during a week. In other words, a simple `GROUP BY` clause over a set of attributes (in this example the four address fields) is not sufficient. Additional rules are needed to distinguish different connections with the same attribute values. For example, we could define a connection as released if for *T* time units no packets have been sent, i.e., packets with the same address quadruple that are sent after such a break belong to a new connection. However, the important aspect of this example is not the particular approach of how connections are identified. Instead, it illustrates a problem that is common to many traffic analysis tasks that cannot be solved by a continuous query in TelegraphCQ. The basic problem is that associations, like flows, sessions etc., must be recognized and each packet must be related to a single association such that statistics for each association can be calculated. For the identification of associations, the address fields in packets are typically used together with certain rules that are depending on the protocol. Before a packet stream is analyzed it is not known which associations with their particular attribute values will be seen. Instead, this information has to be extracted from the data stream. Afterwards, it is possible to assign packets to their associations, based on the attribute values. The traditional solution is to maintain a data structure to store data about associations. Each packet in the data stream can contribute in two ways to the data that is stored in the data structure. First, it is used to analyze whether it belongs to a new association that has to be inserted in the data structure and second, it is used to update the statistic for the particular association it belongs to. With the SQL `SELECT` statement it is not possible to maintain a data structure to store intermediate results. Therefore, the only on-line solution is to analyze the data stream two times, first to identify associations and second to calculate association statistics. Since DSMSs do only allow a single pass over the data stream, it is only possible to perform such a task in a continuous query with subqueries. Since TelegraphCQ does not support subqueries, this type of task cannot be solved on-line in TelegraphCQ.

Task 4. *Which department has used how much bandwidth on the university backbone in the last five minutes?*
To solve this task, a wrapper has to capture all packets from the backbone. A join operation has to be performed between a predefined stored table that contains the IP addresses that are used in the different departments and the stream `p6trace.tcp` from the wrapper. The best solution would be to define for each department the existing IP address range and check with the PostgreSQL operator “>>” which address range contains the IP address of the packet in the data stream. The corresponding DDL statement to create such a table is as follows:

```
CREATE TABLE departments (name varchar(30), prefix cidr,
traffic bigint);
SELECT departments.name, sum(length-hlen-tcp_hlen)
FROM p6trace.tcp, departments
WHERE departments.prefix >> p6trace.tcp.ip_src
GROUP BY departments.name
WINDOW p6trace.tcp ['5 min'];
```

Unfortunately, the current TelegraphCQ prototype produces incorrect results if a “>>” operator is used in a join. However, it works correctly if a “=” operator is used. To solve the task, it is therefore necessary to store all IP numbers that are used by the departments in a stored table:

```
CREATE TABLE departments (name varchar(30), ip_addr cidr,
traffic bigint);

SELECT departments.name, sum(length-hlen-tcp_hlen)
FROM p6trace.tcp, departments
WHERE departments.ip_addr = p6trace.tcp.ip_src
GROUP BY departments.name
WINDOW p6trace.tcp ['5 min'];
```

The major disadvantage of this solution is that all IP addresses must be enumerated and can lead to a very large table.

4.2 TCP Rate Analysis with TelegraphCQ and PostgreSQL

T-RAT [10] aims to identify the reasons for rate limitations, e.g., congestion, receiver buffer constraints, of TCP flows. For simplicity reasons, we focus our description just on the first important steps of the T-RAT algorithm (Fig. 2.a) and how they could be performed in TelegraphCQ and PostgreSQL (Fig. 2.b). The initial step is to identify connections in a TCPdump trace file based on matching IP addresses and port numbers of source and destination. This is not possible on-line. Therefore, in TelegraphCQ a continuous query is used to put the relevant data from a TCPdump wrapper into table T1. The remaining process has to be done off-line in PostgreSQL. First, query Q1 is used to generate table T2 with all connections. Afterwards, query Q2 relates in table T3 all packets to their connections. In order to partition the set of packets in each connection in flights (based on 27 RTT candidates), query Q3 has to join table T3 and the table with the RTT candidates. The join operation in Q3 cannot be solely expressed in SQL and requires an external C function. The next query Q4 requires also an external C function to classify the flights into the different protocols states “slow start”, “congestion avoidance”, and “unknown”, and to find the best fit which indicates the best RTT for the particular connection.

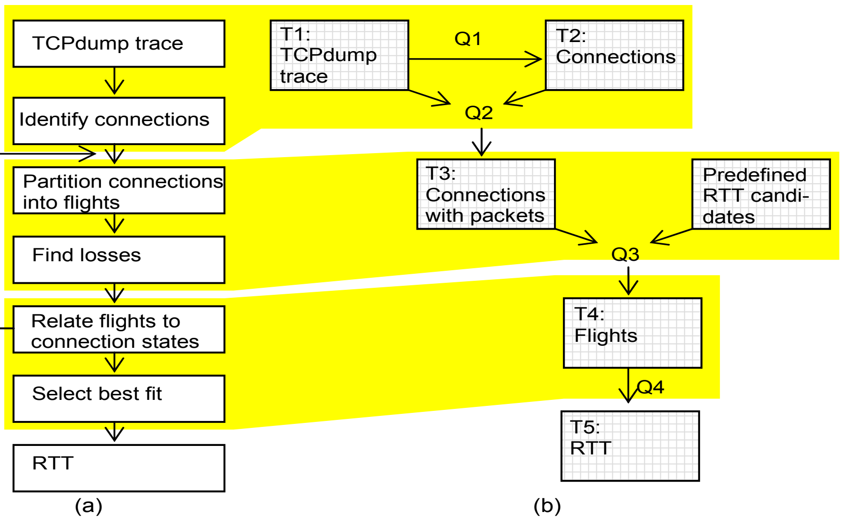

Fig. 2. Structure of the original T-RAT algorithm (a) and its design in Telegraph CQ (b)

The main insights from this design exercise are the following: With TelegraphCQ it is not possible to perform this task in a continuous query, because to identify connections and flights it would be necessary to handle dynamic tables in continuous queries with subqueries. Both are not supported in the current release. The main functionality of T-RAT has to be performed off-line with PostgreSQL. T-RAT is using complex heuristics which cannot be expressed in SQL. However, the extensibility of TelegraphCQ, respectively PostgreSQL, allows to increase the expressiveness of queries with external functions. Thus, T-RAT and other complex analysis can be performed off-line with TelegraphCQ, even if the main functionality is then hidden in external functions instead of SQL statements.

In general, it should be noted that DSMSs are not appropriate for all problems, and that DSMSs require a new way of designing tools for traffic analysis. A developer should especially pay attention to the question which functionality should be performed in a continuous query and which should be performed off-line.

4.3 Performance Evaluation

In order to get a first idea about the performance of TelegraphCQ we performed some experiments in a subnetwork at Institute Eurecom which is based on a 100 Mbit/s Ethernet. The load in this subnetwork is close to zero and its impact on our results can be ignored. We used a simple workload generator to generate a well-defined load for the wrapper and TelegraphCQ. A client streams a fixed amount of data with fixed rates to a server via a TCP connection. This enables us to use the same wrapper and stream definition, i.e., `p6trace.tcp,` as defined in Section 4.1. The size of all TCP packets is equal to the maximum segment size. Due to practical reasons, the wrapper, TelegraphCQ, and the server are running on the same machine, a Pentium 4 machine with a 2Ghz CPU, 524 MB RAM, and a 100 Mb/s Ethernet card.

The basic idea of our performance experiments is to increase the network load until TelegraphCQ is no longer able to handle all data. This load indicates an upper performance bound for TelegraphCQ. We use two mechanisms to recognize whether TelegraphCQ could handle all data or not. First, we log the data that is forwarded from the wrapper to TelegraphCQ, and second, TelegraphCQ itself summarizes how much data it handled. We compare these results with the (amount of) data the workload generator streamed on the network.

Figure 3 shows how the query type and the number of attributes that are handled in the query impact the performance. With a projection, i.e., a simple query that selects a certain number of attributes, TelegraphCQ can handle in maximum 3.6 megabyte per second (MB/s) network traffic for one attribute and 2.7 MB/s for 16 attributes. The upper performance bound for an aggregation is 3.4 MB/s for one attribute (Section 4.1, Task 1) and 2.7 MB/s for four attributes.[2] The most surprising results we got for queries that perform a `JOIN` operation over a stream and a table (Section 4.1 Task 4). A join is normally the most costly operation in a DBMS. We assumed that the performance of a join is not better than the performance of the projection if both generate the same amount of output data. However, in our measurements TelegraphCQ can keep up with 5.8 MB/s of network data for handling one, two, and four attributes in a join when using a small table with ten entries. Further investigations showed that the number of matches between table and stream does impact the performance. The more entries in the table match the attribute value of a stream tuple, the lower the performance. The results in Figure 3 are based on a unique match. The size of the table and

[2] For aggregation and join we increased the number of attributes only to four, because we cannot see any meaningful application that would handle more aggregation or join attributes in these queries.

the position of the matching entry in the table influence the performance. However, even with a table of 100000 entries in which the matching entry is the last one, TelegraphCQ can still handle more than 2 MB/s of network data without loss. In order to verify these preliminary results, more in depth investigations have to be performed. Our results so far can only document that TelegraphCQ is fast enough to perform meaningful network analysis tasks on a commodity PC with a standard Linux configuration (OS release 2.4.18-3) without loosing data up to network loads of 2,5 MB/s.

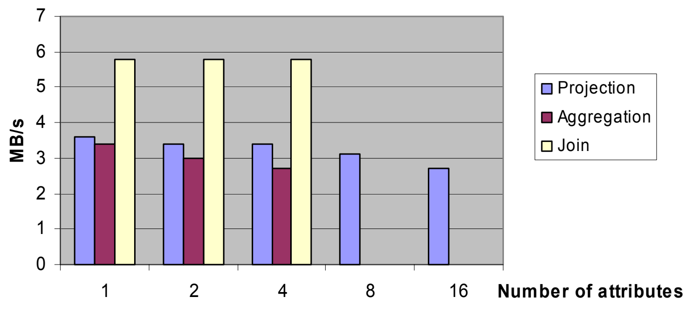

Fig. 3. Upper performance bounds for continuous queries

5 Discussion and Conclusions

Comparing our expectations and requirements with the experiences we gained in this study, we can state that the recent TelegraphCQ prototype is quite useful for many on-line monitoring tasks. The sliding window concept assures that the client gets always the most recent statistics and the system can keep up with relatively high link speeds by running only on commodity hardware. We have shown some preliminary performance measurements and that the performance of TelegraphCQ is influenced by the number of attributes in the output. It should also be noted that the query language can be extended with inbuild C-functions. This feature enables a developer to implement and use in TelegraphCQ complex algorithms that cannot be expressed with SQL. As it is natural for an early prototype, not all features are fully implemented yet. For example, joins between a stream and a table can only compare elements with the "=" operator (i.e., only equi-joins are supported in the current release). Therefore, it is not possible to match prefixes of IP addresses, even if it would be very helpful to identify the origin IP domain of packets. Instead, all possible origin IP addresses we want to compare with have to be stored in the table in advance.

We have identified three restrictions in the design of TelegraphCQ that makes it not suitable as a general tool for traffic analysis in its current stage:

- *Subqueries are not supported*: all tasks that require to identify associations by inspecting the data stream twice, i.e., a simple `GROUP BY` statement over certain attributes is not sufficient, since it cannot be solved on-line.

- *Jumping or tumbling windows are not supported*: a sliding window introduces redundancy, because each tuple contributes multiple times to a sliding window result. To calculate statistics that are correct for time intervals that are longer than a single window, this redundancy has to be removed. However, this can only be done if it is known how and to which intermediate result each packet contributes. However, this knowledge cannot be assumed to be present at the client or an off-line application that is using the set of intermediate results.
- *On-line and off-line handling is not integrated*: TelegraphCQ is designed to forward all results to the user (client). There is no direct way to insert results from a continuous query into a PostgreSQL database. The workaround is to start the system such that its standard output is placed in a file. This file in turn could be later imported into a PostgreSQL database.

These aspects are supported by those DSMS that are targeted for network monitoring, i.e., Tribeca [8] and GigaScope [2]. However, both systems are not available as public domain DSMS. Therefore, we will in future studies focus on STREAM [1] which is promised to be public domain within the next months and supports subqueries. We will also closely follow future releases of TelegraphCQ.

References

1. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems, ACM Symposium on Principles of Database Systems PODS 2002, Dallas, Texas, USA, May 2000
2. Cranor, C., Johnson, T. Spatcheck, O., Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications, ACM SIGMOD 2003, San Diego, California, USA, June 2003
3. Golab, L., Özsu, M. T.: Issues in Data Stream Management, ACM SIGMOD Record, Vol. 32, No. 2, June 2003, pp. 5-14
4. Izal, M., Biersack, E. W., Felber, P. A., Urvoy-Keller, G., Al Hamra, A., Garces-Erice, L.: Dissecting BitTorrent: Five Months in a Torrent's Lifetime, PAM2004, Antibes Juan-les-Pins, France April 2004
5. Krishnamurthy, S., Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Madden, S., Reiss, F., Shah, M. A.: TelegraphCQ: An Architectural Status report, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, March 2003
6. Micheel, J., Braun, H.-W., Graham, I.: Storage and Bandwidth Requirements for Passive Internet Header Traces, Workshop on Network-Related Data Management, in conjunction with ACM SIGMOD/PODS 2001, Santa Barbara, California, USA, May 2001
7. Schleippmann, C.: Design and Implementation of a TCP Rate Analysis Tool. Master Thesis, TU München/Institut Eurecom, August 2003
8. Sullivan, M., Heybey, A.: Tribeca: A System for Managing Large Databases of Network Traffic, Proc. USENIX Annual Technical Conference, New Orleans, USA, June 1998
9. TelegraphCQ: http://telegraph.cs.berkeley.edu/, 2003
10. Zhang, Y., Breslau, L., Paxon, V., Shenker, S.: On the Characteristics and Origins of Internet Flow Rates, ACM SIGCOMM'02, Pittsburg, USA, August 2002