UNIVERSITY OF OSLO
Department of informatics

Design, Implementation, and Evaluation of Network Monitoring Tasks with the STREAM Data Stream Management System

Master thesis
60 credits

Kjetil Helge Hernes

1st May 2006

Preface

The work on the current master thesis commenced spring 2004 and finished spring 2006. I have written this thesis at the research group Distributed Multimedia Systems (DMMS).

I would like to thank my supervisors Professor Vera Goebel and Professor Thomas Plagemann for providing me with the opportunity to write this thesis. I wish to acknowledge them for their advices especially regarding queries and technical difficulties. In addition, I would like to thank Vibeke Søiland for her valuable linguistic assistance. Furthermore, my fiancée Siri and our two lovely children Henrik and Mathea have shown a great understanding and been patience throughout my work. Their love and support has been invaluable. Finally, I would like to thank Jarle Søberg for great companionship and teamwork. Our master theses are closely tied to each other, and I have taken advantage of cooperating with him in many of the areas we have encountered during our work.

Kjetil Helge Hernes

1st May 2006

Abstract

The heavy load and rich variety of data on the Internet has resulted in the need to gain an understanding of the characteristics of the traffic to better plan, develop, and implement new network devices, applications, and protocols. In order to obtain such knowledge, network monitoring is becoming more and more important. However, tools available for network monitoring are restricted to either offline analysis in DBMSs or online analysis through hard-coded continuous queries. Many streaming applications would benefit from a system where network monitoring queries effectively can be inserted, deleted, modified, and processed online in a continuous and real time manner. Data Stream Management System (DSMS) is a promising technology with respect to the needs of network monitoring, because it is designed to meet the above requirements generated by many streaming applications. In the present paper, an experimental analysis of STREAM as a network monitoring tool is performed. STREAM is a general-purpose DSMS and its continuous query language is known as CQL (Continuous Query Language). We investigate whether the current implementation of CQL operators provides us the possibility of expressing a wide-ranged set of network monitoring queries. Furthermore, STREAM's performance is measures by accomplishing several experiments that are processed online over real network traffic.

Results reveal that STREAM can handle network loads up to 30 Mb/s for simple queries, and up to approximately 3 Mb/s complex queries. When queries are executed concurrently, STREAM can handle network loads up to approximately 2.5 Mb/s, strongly depending on the complexity and number of queries.

STREAM provides a well-sized set of operators that provides us the possibility of expressing many types of queries. However, network monitoring queries are restricted by lack of specific network data types and operators. Consequently, these queries are expressed in cumbersome ways. STREAM manages to process network monitoring queries online in a continuous manner, nevertheless at a very limited network load. Thus, the applicability of STREAM as a network monitoring tool is restricted.

Table of Contents

Table of Figures

1. Introduction

1.1 Motivation and Background

In the early years of the Internet, only a small collection of applications, including e-mail, remote login, and file transfer, was employed. However, during more recent years, many new Internet applications have emerged. Examples of such applications are WWW, P2P File Sharing, multimedia streaming, and IP telephony. These new applications have resulted in an Internet that today consists of an enormous and still increasing volume and variety of data. Seeing that the Internet continues to grow rapidly in size and complexity, a detailed understanding of its traffic may become valuable in e.g. design and development of new protocols and applications, traffic engineering and capacity planning, congestion and fault diagnosis, and security analysis. In order to obtain such knowledge, the Internet must be monitored through collecting, measuring, and analysing the data it is carrying.

Since the Internet is continuously in use, much of the network traffic is collected continuously which results in continuous streams of data. A stream of data, or *data stream*, is a potentially infinite sequence of data elements, or tuples, that arrive *online* in a continuous and possibly rapid manner. Several approaches are used to analyse streams of network data. One approach is to capture network traffic and store the data in trace files on disk for later analysis in a Database Management System (DBMS). However, the heavy load on the Internet results in very large and fast-growing databases. Since storage devices have limited capacity, it may not be possible to store such large amounts of data. In addition, DBMSs are not designed for the rapid and continuous arrival of data elements, and they do not directly support the continuous queries that are typical for data stream applications (Babcock et al. 2002). Many such streaming applications would benefit from online analysis and results provided in a real time manner. Due to the insufficiency of traditional DBMSs in providing online continuous query processing, current traffic-management tools are restricted either to offline query processing or to online processing of simple hard-coded continuous queries. These hard-coded continuous queries are often implemented to solve only a very small set of possible network monitoring tasks. In order to solve new tasks, the existing source code, if any, must be extended with additional functionality. However, it is not easy to extend the hard-coded continuous queries, because they are often written in scripting languages e.g. Perl, which make them hard to understand if they are not well documented (Plagemann et al. 2004). A network traffic monitoring system should provide online processing of ad hoc continuous queries over data streams. This would allow network monitoring operators to insert, remove, and modify monitoring queries in a way that supports effective management of the network.

Recent years we have seen several research communities that are motivated from network monitoring, as well as other streaming applications e.g. wireless sensor networks, in order to develop a new technology that supports a more structured approach when solving tasks within these applications. This new technology, the Data Stream Management System (DSMS), is in many ways related to the DBMS technology. However, a DSMS differs from a DBMS in two fundamental ways. Firstly, a DSMS is designed to handle online processing of multiple, continuous, unbounded, and possibly rapid data streams. Secondly, a DSMS is designed to handle continuous queries that produce answers continuously over time. Three different query paradigms, relation-based, object based, and procedural, have been proposed for DSMSs. At present, most research groups attend to the relation-based paradigm, which have languages related to the *declarative* language, SQL (Structured Query Language). In the IEEE Standard Computer Dictionary (Geraci et al. 1991), declarative language is defined as a non-procedural language that permits the user to declare a set of facts and to express queries or problems that use these facts.

In our research group, we are concerned with network monitoring. Since DSMSs are designed to solve challenges imposed by such streaming applications, we investigate how suitable a DSMS is in solving network monitoring tasks online, in a continuous manner.

1.2 Problem Description

Today, there exist several DSMSs implementations e.g. Aurora (Abadi et al. 2003), Gigascope (Cranor et al. 2002), Niagara (Chen et al. 2000), STREAM (Arasu et al. 2004a), and TelegraphCQ (Chandrasekaran et al. 2003), whereof some are available for example as public domain.

As part of a project at Institute Eurécom in France, some of the members in our research group performed a case study by using TelegraphCQ for traffic analysis. They concluded that TelegraphCQ is relatively useful for many online monitoring tasks (Plagemann et al. 2004). However, several limitations regarding the TelegraphCQ prototype available at the time, were identified. Conclusively, they found that it was not suitable as a general tool for network analysis (Plagemann et al. 2004). Consequently, we want to perform a similar, however, more comprehensive study where the performance of another DSMS network monitoring tool is being analysed. Initially, as the present project commenced, three public domain DSMSs were available. These were Niagara, STREAM, and TelegraphCQ. However, the research group developing Niagara was no longer active and another master thesis is focusing on TelegraphCQ. Consequently, we choose to focus on STREAM, which is developed at Stanford University, California.

STREAM is developed as a general purpose DSMS. However, our focus is on network monitoring and the following problem description will be discussed:

Evaluate the applicability of STREAM as a tool for online and continuous network monitoring.

We evaluate the applicability by, firstly, investigating the expressiveness of network monitoring queries with the current CQL implementation in STREAM and, secondly, by measuring STREAM's performance while processing queries online over a stream of tuples captured from live network traffic. This process of evaluating STREAM's applicability consists of three major steps. Firstly, we design queries solving several network monitoring tasks in order to investigate the expressiveness offered by the implementation of operators in the current STREAM prototype. Secondly, we implement an experimental setup in which we generate network traffic. The experiment setup includes functionality for capturing network packets and pushing a stream of tuples consisting of the packets' header values into STREAM. Thirdly, we complete several experiments that are designed to measure the performance of STREAM as it processes continuous network monitoring queries over the stream of tuples produced in the experimental setup.

- *Query design*: When designing queries, we are concerned with the expressiveness of CQL, which represents STREAM's query language. We investigate this expressiveness based on the collection of operators implemented in the current STREAM prototype in the context of designing queries that solve a wide-ranged set of network monitoring tasks.

- *System implementation*: We implement a system that enables packet capturing, because STREAM is not implemented to obtain data from live sources. In addition to packet capturing, this system includes the extraction of packet header values, as well as the transformation of these values into tuples of a representation accepted by STREAM. Furthermore, we implement an experimental setup in order to generate network traffic in a controlled manner for our performance evaluation. A collection of small tests is performed in order to confirm the correctness of this setup.

- *Performance evaluation*: We measure STREAM's performance by completing several experiments. We define a workload and a set of factors that decide the scope of the environment for these experiments. In addition, we define a set of metrics in order to enumerate the measurement of STREAM's performance.

The performance of STREAM is measured as a *black box*. Consequently, we only measure the performance in terms of its input and output characteristics. The system is not evaluated in terms of program structure, nor do we add extra query processing or monitoring functionality. However, we investigate the source code in order to gain a better understanding of the internal mechanisms of STREAM.

The terms "bits" and "bytes" are used repeatedly throughout this thesis. To avoid confusions, the use of the terms are clarified below. We write "bits" and "bytes" for these terms, however, "M" is employed as an abbreviation for "mega". When writing "megabits" and "megabytes" we use the abbreviations "Mb" and MB", respectively. "Megabits per second" is applied as the metric for network load and is written as "Mb/s". The binary notion on "mega" is utilised, thus, one Mb is 1048576 bits and one MB is 1048574 bytes. One byte is equivalent to eight bits.

1.3 Outline

Jarle Søberg, another master student, writes a thesis (Søberg 2006) with a similar problem description and content as the current thesis. However, Jarle Søberg is accomplishing a performance evaluation of TelgraphCQ. The work on our theses is in many ways similar. We have developed a common experimental setup, and many of the network monitoring tasks we solve are equal. Since we both are investigating DSMSs, we have similar motivation and background. Consequently, we collaborate in the writing of Section 2 and Section 3, which together with Section 4 constitute the theoretical background of the theses. These sections are similar in the two theses, with small adjustments to fit the structure of each thesis.

In Section 2, some of the streaming applications that serve as motivation for the development of DSMSs are described. In addition, we analyse requirements that these applications impose on the DSMSs.

Section 3 presents an introduction to important concepts in the DSMS technology. We give a more thorough description of the most important issues and challenges imposed by this technology and describe DSMSs in general.

In Section 4, we describe STREAM in particular by presenting STREAM's query language, CQL, along with an architectural overview of the system. In addition, important concepts within STREAM will be discussed. Finally, we describe how to use the system.

We start Section 5 by defining an input stream that is used throughout the rest of the thesis. Moreover, we discuss some challenges concerning what data type to choose when representing the different attributes in this stream. The main contribution to this section is the design of queries that solve network monitoring tasks.

In the following section, Section 6, we describe how we make live data sources available to STREAM and how we implement a packet capturing tool. In addition, the experimental setup is described. Finally, some tests are performed in order to assure the correctness of these implementations.

Section 7 is concerned with presenting the performance evaluation. Initially, we define factors, workload, and metrics, followed by the design and presentation of results for the different experiments. At the end of this section, we discuss and compare the experiments.

In the final section, Section 8, we draw some conclusions based on our results, summarise the work on the thesis, and give some critical assessments. In addition, open problems and future work will be discussed.

Finally, an appendix that describing the organisation of the DVD-ROM attached to this thesis will be included.

2. Streaming Applications

Traditional databases have been utilised in applications that require persistent data storage and complex querying. Usually, a database consists of a set of records, with insertions, updates, and deletions occurring less frequently than queries. The database system executes the query when it is posed and the answer reflects the current state of the database. However, during the recent years we have seen an emergence of applications that do not fit the data model and querying paradigm of traditional databases (Golab et al. 2003). In these applications, data is better modelled as transient *data streams* than as persistent relations. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, Web applications, manufacturing, and sensor networks. Individual data items in a data stream may be relational tuples e.g. network measurements, call records, Web page visits, and sensor readings (Babcock et al. 2002). In the *data stream model*, some or all of the input data that are to be operated on, are not available for random access from disk or memory, but rather arrive as one or more *continuous data streams*. Data streams differ from the conventionally stored relation in several ways (Babcock et al. 2002; Plagemann et al. 2004):

- The data elements in the stream arrive online and remain only for a limited time in memory. Consequently, the data elements must be handled before the buffer is overwritten by new incoming data elements.

- The system has no control over the order in which data elements arrive in order to be processed, either within a data stream or across data streams.

- Data streams are potentially unbounded in size and may be regarded as open-ended relations.
- Once an element from a data stream has been processed it cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data stream.
- A data stream is *append-only*, which means it only consists of insertions, and not any deletions or updates.

To integrate data collection and processing, and to enable online (as well as offline) processing, several research communities have proposed the use of DSMSs for deploying these new streaming applications. Instead of processing queries over a persistent set of data that is stored in advance on disk, a DSMS processes *continuous queries* over the arriving data elements. Continuous queries are evaluated continuously as the data streams arrive. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrive, or they may be produced as data streams themselves. In Section 3, we describe the DSMS technology extensively. In Section 3.4.1, we give a more thorough discussion of continuous queries.

Streaming applications may be divided into two different categories: *pull-based* and *push-based* applications. In pull-based applications, data is “pulled” from the data sources into the system, as in traditional database systems. In push-based applications, data elements are “pushed” from the data source into the system. In the current thesis, we consider network monitoring, which is a push-based application domain. Consequently, we emphasise the discussion of the push-based domains, with network monitoring in particular. The main pull-based streaming application domain is sensor network, which is the only pull-based domain that will be discussed. Pull-based and push-based application domains generate a set of requirements that a streaming application system e.g. a DSMS, should accommodate. At the end of this section, we perform an analysis of such requirements.

2.1 Sensor Networks

Traditional sensors deployed throughout buildings, labs, and equipment, are passive devices that simply transmit signals based on some environmental parameter. Such nodes are for example connected to a local area network (LAN) and attached to permanent power sources. However, recent advances in computing technology have led to the production of a new class of devices: the wireless, battery-powered, computing sensors. These new devices are active computers, capable of not only sampling real-world phenomena, but also filtering, sharing, and combining sensor readings with each other and nearby Internet-equipped end-points. Such sensors may be adjusted in order to allow a suitable degree of precision, for example reporting every second or every fifth second. The sensor nodes communicate via wireless multi-hop radio powered by small batteries (Gehrke et al. 2004; Madden et al. 2002; Yao et al. 2003) and are made of four basic components: a sensing unit, which is usually composed of sensors and analogue to digital converters (ADCs), power units, a transceiver unit, and a processing unit. When describing sensor networks, we only consider networks consisting of the wireless sensor type. In Figure 1 below, we give an example of a sensor network. We see that the sensors communicate with each other and/or a central node or access point, which is labelled AP in the figure. The dotted arrows show communication links. The sensors pull data (e.g. light or noise) from the environment based on the functionality of their sensing device, and send the data through the network back to a central node for querying and data analysis. The transmission of data from sensor to sensor towards the central node generates a data stream consisting of sensor readings with the elements arriving at a constant rate. However, this data transmission is extremely expensive for sensor networks since communication using the wireless medium consumes a considerable amount of energy (Yao et al. 2003). Since sensors have the ability to perform local computation, part of the computation may be moved from the central node and pushed into the sensor network. Then sensors can aggregate records, or eliminate irrelevant records. Compared to traditional centralised data extraction and analysis, in-network processing can reduce energy consumption and bandwidth usage by replacing more expensive communication operations with relatively cheaper computation operations. This may extend the lifetime of the sensor network significantly (Yao et al. 2003). Based on this structure, sensor networks can be applied on a wide range of applications.

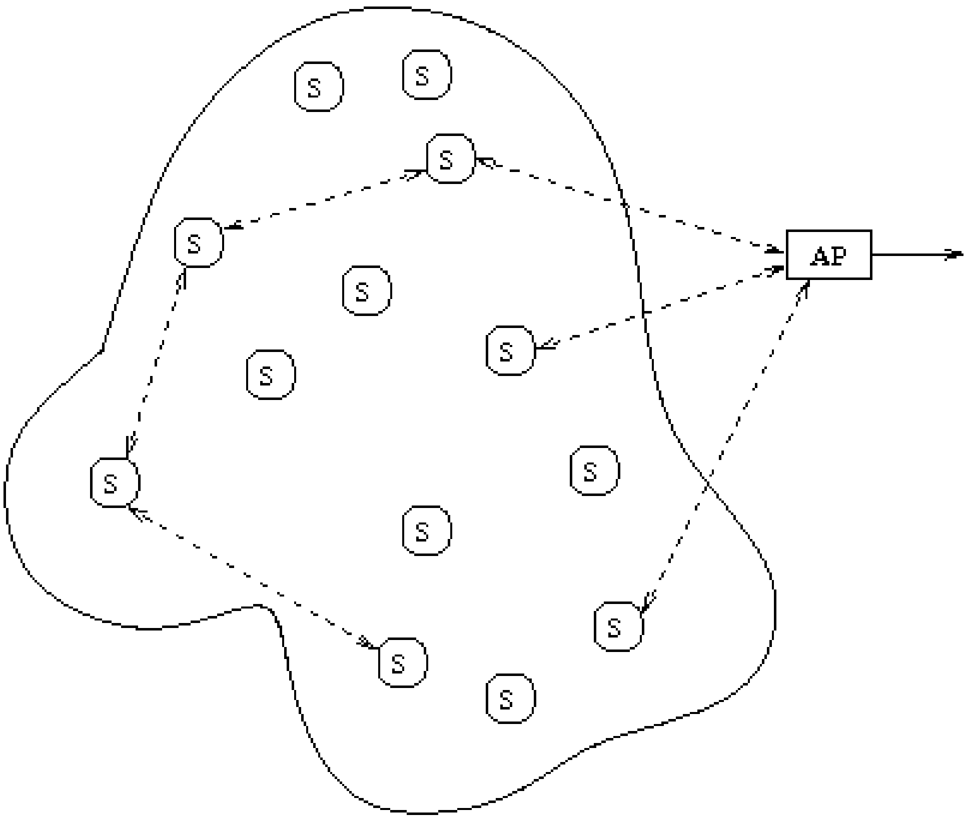


Figure 1. An overview of sensors in a sensor network

2.1.1 Sensor Network Applications

It is predicted that we in the future will see a more extended use of sensor networks, because sensors become smaller and more inexpensive (Madden et al. 2002). However, already today there are many sensor network applications. Among these are military applications, which through military funding gave birth to many research projects within the field of sensor networks in the early 1980s (Chong et al. 2003). In the following, we give examples of other sensor network applications.

In a national park, sensor networks can cover large areas over large periods. They can capture microclimates, report unusual seasonal events, and monitor animal behaviour. For instance, as part of a project at UC Berkeley (Gehrke et al. 2004), it was used small sensors to investigate the microclimate at the redwood trees in the UC Berkeley botanical garden. This network played an important role in assisting the botanists in their research and data collection.

Sensor networks can monitor roads for accidents and traffic hotspots, and warn approaching drivers about the incidents. In such cases, sensor networks can help in diverting traffic, thus increasing transport capacity. Other applications may be to manage road tolling, parking spaces and to detect illegal driving (Madden et al. 2002). For example, sensors may be placed in streets where there is heavy traffic in order to investigate the driving pattern by registering the number of cars passing by.

Sensor networks can assist in identifying early signs of fire in forests by helping fire fighters to predict the direction in which the fire is likely to expand. Sensor networks may also assist in rescue operations by locating victims or members of the rescue team.

2.1.2 Limitations of Sensors

Though many new applications have risen following the development of wireless sensors, these sensors also introduce limitations, which constrain their applicability. We list some of the main limitations here:

- **Power** is the defining limit of sensor nodes: it is always possible to use a faster processor or a more powerful radio, but these consume more electricity, which is often not available. Thus, energy conservation is an essential system design consideration of any sensor network application. An example of a sensor is the Berkeley MICA mote (Yao et al. 2003). The mote is powered by two AA batteries, which provide about 2000 mAh, powering the mote for approximately one year in the idle state and for one week under full load.

- **Communication:** The wireless network connecting the sensor nodes has limited bandwidth (Madden et al. 2002; Yao et al. 2003), latency with high variance, high packet drop rate, and usually provides only a very limited quality of service (Yao et al. 2003).

- **Computation:** Limited computing power restricts algorithmic complexity available to a sensor. In addition, scarce memory resources restrict the amount of intermediate results a sensor can store (Yao et al. 2003). Recently, small operating system e.g. TinyOS (Culler et al. 2001), and small database systems e.g. TinyDB (Madden et al. 2005), have been developed in order to handle these computational limitations.

- **Routing:** For wireless networks, some of the nodes may be mobile in the sense that they are attached to moving objects. In such cases, one has to use special routing algorithms to identify the location of the sensor, maintain a network topology, and verify that the sensor is working as planned. An example of such a routing algorithm is the optimised link state routing protocol (OLSR) (Clausen et al. 2003), which is developed for mobile ad hoc networks (MANET) (Corson et al. 1999).

2.2 Push-Based Applications

In push-based applications, the system cannot control the rate at which data elements arrive. These applications are concerned with data stream that are often characterised by bursts and heavy load. We discuss three push-based application domains in this section: network monitoring, transaction logs, and financial tickers. The discussion of network monitoring is largely emphasised.

2.2.1 Network Monitoring

In 2002, the Internet consisted of nearly 12 000 Autonomous Systems (ASes). Each AS is a collection of routers and links managed by a single institution, such as a company, university, or Internet Service Provider (ISP) (Grossglauser et al. 2002). The evolution of the Internet is closely tied to detailed understanding of its traffic. Moreover, tools to analyse Internet traffic are becoming more and more important as the Internet continues to grow rapidly in size as well as in complexity. Hence, operators of large networks and providers of network services need to monitor their network by measuring and analysing the network traffic flowing through their systems. Network monitoring can provide a valuable insight into the dynamics of network traffic protocols, traffic engineering and capacity planning, congestion and fault diagnosis, and security analysis (Hussain et al. 2005). Many monitoring applications are complex (e.g. reconstruct TCP/IP sessions), operate over huge volumes of data (e.g. Gigabits and higher speed links), and have real-time reporting requirements e.g. to raise performance or intrusion alerts (Cranor et al. 2002). In a network, one may collect data at several locations (e.g. hosts and routers) both inside the network as well as at the edges. Such data includes (Babu et al. 2001):

- Data from network packets and flow traces. Such data may contain information like header fields and packet data. Network packets can be captured by passively listening to the traffic on the network.

- Data obtained by measuring packet delay, loss, and throughput. Such data can be obtained by measuring the behaviour of packets that actively are sent through the network.

- Router forwarding tables and configuration data. The routers in the network send data packets to each other describing network characteristics and routing information. Such data can be used to get a total overview of the traffic at several routers. An example is the Simple Network Management Protocol (SNMP) (Case et al. 1990). Data from this protocol is used to communicate network information between the gateways and the network administrators.

Broadly, network traffic monitoring can be divided into three tasks:

1. Collecting the data e.g. router configuration data.

2. Measuring the collected data e.g. to obtain statistics from the collected data.
3. Analysing the measured data e.g. to characterise and model traffic in various layers.

In this section, we focus on task two and three. Firstly, we consider network traffic measurement, whereas secondly, network traffic analysis will be discussed.

Network Traffic Measurement

Network traffic measurements play a crucial role in providing operators with a detailed view of the state of their networks. These measurements are conducted on a continuous basis and the results are compiled into reports for management that are used in management decisions on various time scales. Traffic measurements are divided into two different techniques; *passive* and *active* measurements. When using the passive technique, one simply observes and records the traffic as it passes by. This approach measures real traffic, is useful for characterising the Internet traffic, and does not disturb the network traffic by adding extra load. However, one does not have full control over the measurement process (Siekkinen 2006). When using the active technique, one injects packets into the network, monitors them, and measures the services obtained. This technique is useful for inferring the network characteristics, and one obtains complete control over the measured traffic. However, this technique may disturb network traffic by adding extra load (Siekkinen 2006). In addition to the active and passive techniques, we may divide network measurement into two different approaches; *online* and *offline* measurements. With online measurements the traffic is captured and the data is measured in a real-time manner, whereas with offline measurements traffic is captured into trace files for later measurements.

Examples of traffic measurement tasks, adapted from the STREAM query repository (Anonymous 2002), are:

- For each source IP address and each five-minute interval, count the number of bytes and number of packets resulting from HTTP requests.

- Find the source-destination pairs in the top five percentile in terms of total traffic in the past 20 minutes over a backbone link B.
- Generate the flows in the packet stream, and for each flow, output the source and destination addresses, the number of packets constituting the flow, and the length of the flow.

Other examples are:

- For each source IP address, count the number of active flows from that address in each five-minute interval. A flow may be defined as all packets that have the same source and destination IP addresses, where successive packets have an inter-arrival time less than 30 seconds.
- Maintain the fraction of packets on a particular backbone link B generated by a particular customer network C in the past hour.

Network Traffic Analysis

In network traffic analysis, we use the measurements to maintain network state, to detect causes of problems in the networks, and in capacity planning and optimisation. Broadly, network traffic analysis can be divided into three different categories: Traffic characterisation and modelling, network characterisation and modelling, and anomaly detection (Siekkinen 2006).

Traffic Characterisation and Modelling

Network traffic packets may be recognised based on the characteristics of the protocols they use at different layers. At the application layer the traffic may be related to e.g. peer-to-peer file-sharing applications and Skype. At the transport layer traffic is typical related to TCP and UDP.

After the widespread usage of peer-to-peer (P2P) networking during the late 1990s, P2P applications have multiplied. Their diffusion and adoption are witnessed by the fact that P2P traffic accounts for a significant fraction of Internet traffic (Spognardi et al. 2005). Furthermore, there are concerns regarding the use of these applications, particularly when they are employed to share copyright protected material. In addition, many ISPs are reluctant to let customers consume bandwidth in the file sharing operations, which is a part of P2P applications. It is important to gain a deeper understanding of the characteristics of P2P traffic, because it accounts for a significant part of the Internet traffic. Such knowledge may be valuable in further development of P2P applications or new protocols. However, P2P traffic must be identified before it may be measured and analysed. Identifying P2P traffic may not be easy, because it for instance may camouflage by using TCP ports that are not well known.

Another protocol that it is important to understand in a best possible manner is TCP, because it carries over 90 % of the network load in the Internet (Siekkinen 2006). As for P2P applications, TCP traffic must be identified before it can be measured and analysed. Identifying TCP packets is straightforward, because the IP header provides this information in one of its fields. However, as indicated in Section 5.2.4, recognising TCP *connections* is not a trivial task, particularly not online.

Network Characterisation and Modelling

Managing a large network is a complex task, which may be conducted by a group of human operators. These operators track the characteristics of the network to detect equipment failure and shifts in traffic load. Network characteristics and modelling may be based on parameters such as e.g. topology, utilisation, packet delay, and packet loss. By joining SNMP data and/or configuration data from different network elements, it is possible to maintain network topology, and by aggregating packet traces or SNMP data, it is possible to maintain statistics of link and router utilisation. Packet loss, per-hop and end-to-end delays, and network throughput are measured by either joining packet traces collected from multiple points in the network, or using a dedicated system that generates network traffic to measure these parameters (Babu et al. 2001).

Traffic engineering is concerned with performance optimisation of traffic-handling in operational networks. When optimising performance, it is important to minimise over

-utilisation of capacity when other capacities are available within the network. Updated information on network characteristics is important in order to detect problems with network traffic. After detecting and troubleshooting a problem, operators may change the configuration of the equipment to improve utilisation of the network resources and the performance experienced by end users. An example of a performance problem is *link congestion*. A link may be congested because of an increase in demand between some set of source-destination pairs or a failed link or router in a network causing changes in routes. One way to detect link congestion is to calculate utilisation statistics from SNMP.

For an ISP it is important to have knowledge regarding characteristics of bandwidth consumption in order to make proper allocation of resources or to decide where and when to install new equipment. Examples of decisions to make are where to put the next backbone router, when to upgrade a peering link to higher capacity, and whether to install a caching proxy for cable modems.

Anomaly Detection

The widespread usage of the information-sharing possibilities provided by the Internet has revolutionised our society by enabling us to communicate easily with people around the world, and to access and provide a large variety of information-based services. However, this success has also enabled the use of Internet in ways that are considered hostile, and spam, viruses, worms, and denial-of-service attacks (DoS) are today well known terms. As the number of network-based attacks increase, and the variety and sophistication in these attacks grow, early detection of potential attacks will become crucial in reducing the impact of these attacks. We show some examples of anomalous activity by describing denial of service (DoS), worms and viruses, and the probing for vulnerability.

A DoS is characterised by an explicit attempt by attackers to prevent clients from using a service. DoS have been among the most common form of Internet attacks. The basic form of a DoS is to consume scarce computer and network resources, such as kernel data structures, CPU time, memory and disk space, and network bandwidth

(Johnson et al. 2005). An example of a DoS is the TCP SYN flood attack, which exploits the three-way handshake used to establish a TCP connection (Johnson et al. 2005). In a normal scenario, a sender initiates a TCP connection by sending a SYN packet i.e. a packet having the SYN bit set. The receiver responds with a SYN/ACK packet, and the sender completes the three-way handshake by sending an ACK packet. Following the sending of the SYN/ACK packet, the receiver allocates connection resources (kernel and data structure) to remember the pending connection for a pre-specified amount of time. The attack occurs when the attacker repeatedly sends SYN packets, typically with different source addresses, causing the receiver to deplete its connection resources, preventing service to other users. In principle, the attack can be identified by measuring and analysing the number of SYN packets for which a SYN/ACK packet is sent, but no correlating ACK packet is seen within a given delay.

A *worm* is a self-propagating malicious code, which exploits vulnerabilities in the underlying operating system to inflict its damage, and to replicate and propagate itself (Johnson et al. 2005). A *virus*, on the other hand, relies on user actions for its propagation, and hence tends to spread slowly. Payload and specific mechanism of propagation may identify known worms. For example, activity of the Slammer worm is identifiable in a network by the presence of 376 bytes UDP packets, destined for port 1434/UDP of SQL Server (Johnson et al. 2005).

We see that attacks exploit known vulnerabilities in services. A typical precursor to attacks is the identification of machines that have specific services available, and hence can potentially be exploited. This takes the form of an attacker probing for open ports on a set of host machines. To determine if a port is open, an attacker sends a packet to a host attempting to connect to the specific port. If the target host is listening on that port, it will respond by opening a connection with the attacker. This implies that during the probing phase, the attacker would not spoof the IP source address (Johnson et al. 2005), meaning that such anomalous activity can be detected by measuring and analysing the number of distinct <destination IP, destination port> pairs with the same source IP address.

2.2.2 Transaction Logs

Massive transaction streams introduce a number of opportunities for data mining techniques. Examples of transactions are calls on a telephone network, commercial credit card purchases, stock market trades, and HTTP requests to a Web server (Cortes et al. 2000). The goal is to find interesting customer behaviour patterns, identify suspicious spending behaviour that could indicate fraud, and forecast future data values (Golab et al. 2003). A transactional data stream is a sequence of records that logs interactions between entities. For example, a stream of credit card transactions contains records of purchases by consumers from merchants. Data mining techniques are needed to exploit such transactional data streams since these streams contain a huge volume of simple records, any one of which is rather uninformative unless it is part of a total overview (Cortes et al. 2000). However, when the records related to a single entity are aggregated over time, the aggregate can yield a detailed picture of evolving behaviour, in effect capturing the "signature" of that entity. A signature for a phone number might contain directly measurable features such as when most telephone calls are placed from that number, to what regions the calls are placed, and when the last call was placed. Queries investigating these matters may be quite similar to those detecting anomalous activity in the Internet. It might also contain derived information such as the degree to which the calling pattern from the number is "business-like" (Cortes et al. 2000). Such information is useful for target marketing and for developing new service offerings. Other examples of transaction log analysing tasks, which are adapted from Golab et al. (2003), are:

- Find all pages on a particular Web server that have been accessed in the last fifteen minutes with a rate that is at least 40% greater than the running daily average.

- Examine server logs and re-route users to backup servers if the primary servers are overloaded.

Other examples are:

- Track mobile phone records and for each mobile phone number, determine the number of
    1. Distinct base stations used during one telephone call.
    2. Bytes transferred in order to open Web pages using the wireless application protocol (WAP) (Montenegro et al. 2000).
    3. Bytes transferred in order to download e.g. ring tones, games, and wallpapers.

2.2.3 Financial Tickers

In the United States, up to 100 000 quotes and trades (ticks) are generated every second (Zhu et al. 2002). This results in a stream of stock market transactions, which consist of buy or sell orders for particular companies from individual investors. Online analysis of streams of financial tickers might help a stock market trader to discover correlations, identify trends and arbitrage opportunities, and forecast future values. Traderbot, a typical Web-based financial ticker, allows its users to pose queries such as the following (Golab et al. 2003):

- High Volatility with Recent Volume Surge: Find all stocks priced between $20 and $200, where the range between the high tick and the low tick over the past 30 minutes is greater than three percent of the last price, and where in the last five minutes the average volume has surged by more than 300%.
- NASDAQ Large Cap Gainers: Find all NASDAQ stocks trading above their 200-day moving average with a market cap greater than $5 Billion that have

gained in price today between two and ten percent since the opening, and are within two percent of today's high.

- Trading Near 52-week High on Higher Volume: Find all stocks whose prices are within two percent of their respective 52-week highs that trade at least one million shares per day.

2.3 Requirements Analysis

In this section, we analyse the requirements that are imposed by the application domains mentioned in the previous subsections. Firstly, we consider the common requirements of these applications. Secondly, we discuss the requirements raised by sensor networks and network monitoring. Recall that there is a major difference between standard database sources, and the data sources for the network monitoring applications. Network monitoring applications have to handle data streams i.e. data elements, or tuples, that are continuously produced and pushed into the system. One important requirement raised by streaming applications is that queries over such data streams need to be processed immediately, in real-time. This because it is expensive to save such large amounts of data to disk and much of the data may not be of interest later. Moreover, the streams represent real world events that need to be responded to. Generally, all streaming applications need a system to handle large amounts of arriving data packets. If not all the data is considered interesting, it needs functionality for choosing only the packets that are most important with respect to the application. In cases of much important data tuples, the system some times has to aggregate on the streams in such a way that only the most representative tuples or averages of the data results are displayed. In addition, the systems have to respond quickly to sudden changes in the data streams and register or output these changes. In handling this, a set of general requirements for streaming applications emerge. In the following, we list these requirements, many of which are collected from (Golab et al. 2003).

- *Continuous queries*: To analyse a large range of behaviours attached to the different applications, one would need to collect data on an ongoing basis rather than as a one-time event. Hence, it is required that a DSMS is capable of processing data in a continuous manner. This means that the query has to be started and stopped explicitly by a user or by a system. If not stopped, it is assumed to run infinitely.

- *Projection:* To reduce the size of queues in memory and in turn improve memory utilisation, it is required that a DSMS supports projection i.e. choosing only a subset of the attributes in a relational tuple.

- *Selection*: All DSMSs require support for complex filtering. The selection should manage to fetch only data having certain values, such that much data can be excluded from further processing at an early stage. As an optimising factor, it should be possible to push both selections and projections as close to the source data stream as possible.

- *Joins*: In order to perform a wide range of analysing tasks, a DSMS should include support for joins between multiple streams and joins between streams and stored relations. By supporting this requirement, the DSMS may analyse the data to find patterns that depend on correlations between many streams and relations.

- *Aggregation*: By supporting aggregations, the DSMS may attach statistics to application dependent patterns that it recognises within streams and/or relations. The aggregations, which calculate sums, maximums, minimums, counts, and averages, may also assist in obtaining an overview of the data values in the stream.

- *Windowed queries:* Many operators (e.g. aggregating operators) are *blocking* i.e. the operator must see all input data before it can produce any output. However, data streams are considered infinite. Therefore, the DSMSs have to support some type of partitioning over the streams, such that blocking operators may process data within such partitions or windows. If windowing is not supported, blocking queries can never be performed correctly, since the DSMS needs to see all tuples of the stream in order to compare them.

- *Processing multiple queries*: In many scenarios, multiple users pose similar queries over the same data streams. Since streams are append-only, there is no reason that a particular data item should not be shared across many queries (Madden et al. 2002). Hence, a DSMS should support multiple, concurrent queries.

- *Sub-queries*: To analyse characteristics of an application, the DSMS should be able to perform complex queries to identify mechanisms within the application. In order to perform such complex queries (e.g. reconstructing TCP connections), support for sub-queries is required. Sub-queries may appear in several different query clauses e.g. in selections and in more complex projections.

- *Nested aggregation*: Complex aggregates, including nested aggregates (e.g. comparing a minimum with a running average) may be needed to compute trends in the data sets. A nested aggregate is an expression $agg_n(agg_{n-1}(...agg_0(X)...))$, where each agg_i is an aggregate function and $n \geq 1$ (Johnson et al. 1999). Nested aggregations must be calculated continuously within windows.

- *Multiplexing* and *demultiplexing*: This requirement can be viewed as the group-by aggregation and union set operator, respectively. The multiplexing and demultiplexing can be used to decompose and merge logical streams, depending on the answers required.

- *Frequent item queries*: These are known as top-k or threshold queries, depending on the cut-off condition. Thus, the DSMS only query for items that appear frequently, and may be of greater importance than other items. This may, in addition, be part of the selection such that for instance only the tenth tuple, or values over a certain threshold are selected.

- *Stream mining*: Operations such as pattern matching, similarity searching, and forecasting are required for on-line mining of streaming data. The mining thus relates to a more experience-based and intelligent way of running the queries. If, for example, a query in a weather monitoring network is told to report data only when it is rainy, it might have to compare its input data to historical data and other observations to get assistance in the decision process.

- *Adaptive query processing*: A fundamental challenge in many streaming applications is that conditions (e.g. data values) may vary significantly over time. Since queries in these systems are usually long running, or continuous, it is important to consider adaptive approaches to query processing. Without adaptivity, performance may drop drastically as stream data and arrival characteristics, query loads, and system conditions change over time (Babu et al. 2004).

The above list is used throughout the thesis as a reference. We show how the DSMSs and STREAM implement these requirements. Following is a short additional list of requirements for the pull- and push-based data stream models, exemplified with sensor networks and network monitoring, respectively.

2.3.1 Sensor Networks

The main limitation in sensor networks is based on power consumption, which provides some additional requirements to the application. As sensors are part of a pull-based stream model, the queries are required to specify the pull interval i.e. identifying when to sample data from the environment. This interval has to be relative to the power consumption. It is also required that sensor networks distribute queries among the nodes to reduce the amount of data, because sending data through wireless links consumes much power. This means that each node plays a part in the total query processing. For example, the node may perform some simple aggregations on the data before sending it to another node.

In some sensor network monitoring applications, it may be necessary with a large-scale deployment of sensor nodes. A large number of nodes may even require more scalability in cases where additional nodes may be inserted into or removed from the network.

2.3.2 Network Monitoring

Since networks need to be running all the time, much of the network traffic data is collected continuously and results in very large and fast-growing databases. However, it may not be possible to save such large amount of data to disk, and queries over such data streams need to be processed online. In addition, in many applications data may arrive in bursts, with unpredictable peaks during which the load may exceed system resources. Consequently, it is required that the DSMS provides good approximation techniques in order to keep the query answer as correct as possible. An example of an approximation technique is *load shedding* i.e. dropping elements from query plans and saving the CPU time that would be required to process them to completion (Arasu et al. 2004a).

The load on a network usually consists of traffic belonging to many network protocols. Thus, another requirement imposed by network monitoring is that packets belonging to different protocols should be processed by the DSMS. Additionally, it is important that the DSMS supports the different data operators that may be required in a packet header. An example is the IPv4 address, which consists of four numbers separated by dots. Moreover, a data packet may offer complexity with regard to a varying number of header fields. For example, both the IPv4 (Postel 1981a) (henceforth IP) and TCP (Postel 1981b) headers have option fields, which contain optional information. A DSMS should provide functionality for supporting these variations. Another example is the extension headers in IPv6, which amongst others contain routing information.

Some of the network protocols tend to be complex. For instance the TCP standard specifies how two nodes should act when they establish and close down a connection

(Postel 1981b). Thus, when measuring and analysing network and protocol behaviour, it is required that the DSMS manages to reflect protocol states in both the network and the network nodes. Hence, the declarative language provided by a DSMS should provide a wide range of operators in order to express queries that may be used when monitoring protocol behaviour.

3. Data Stream Management Systems

3.1 Introduction

The current section describes and discusses some of the main issues in DSMSs. A DSMS is a system that poses queries on a stream of data. Based on the requirements in Section 2.3, we reveal how these are designed in the DSMSs. Throughout the thesis we use the following definition of a stream adapted from Arasu et al. (2004b). Given the discrete ordered time domain T:

Definition 1 (Stream) A stream S is a possibly infinite bag (multi-set) of *elements* $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in \mathrm{T}$ is the *timestamp* of the element. There is a finite but unbounded number of stream elements for any given timestamp $\tau \in \mathrm{T}$.

As mentioned in Section 2, several applications motivate for a system that is, readily and in real-time, able to extract relevant information from data streams. Examples of such applications are sensor networks, financial tickers, transaction logs, and network monitoring. As stated in Section 1, the present thesis focuses on network monitoring and therefore, the description of the DSMS is concentrated around this application.

DSMSs are often compared to database management systems (DBMS) since both deal with the querying of data. The following section gives a short review of the DBMS's main issues, terms, and characteristics. The differences are shown by comparing the two systems. We further discuss several of the DSMS requirements listed in Section 2.3.

3.2 Database Management Systems

One of the DBMSs' most important tasks is to generate a query plan that obtains data such as network packet traces from the storage device (e.g. hard disk) as efficiently and quickly as possible (Garcia-Molina et al. 2002). The data is mostly represented using *relations* (or tables). Based on Garcia-Molina et al. (2002), a relation is described as a two-dimensional array for representing data. In the following, some terms that are used to describe certain elements in the DBMS's relations are introduced. These terms are introduced since as they describe the equivalent elements (e.g. sequences of data) in the DSMS. We focus on the relational representation that organises the data in columns and rows. A row (henceforth *tuple*) consists of several values, one for each column (henceforth *attribute*). The tuple has a *fixed* and predefined amount of attributes. Figure 2 below illustrates how these terms are used. The relations are stored in *blocks* on the storage device and may be *joined* by pairing those tuples that in some way match each other. By join, we mean *natural join* or *theta join*. A natural join pair are only those tuples that agree in whatever attributes are common to the *schemas* of the relations. Theta joins are produced by taking the Cartesian product, and then selecting from the product only those tuples that satisfy the given conditions (Garcia-Molina et al. 2002). A DBMS is recognised by a number of characteristics (Babcock et al. 2002):

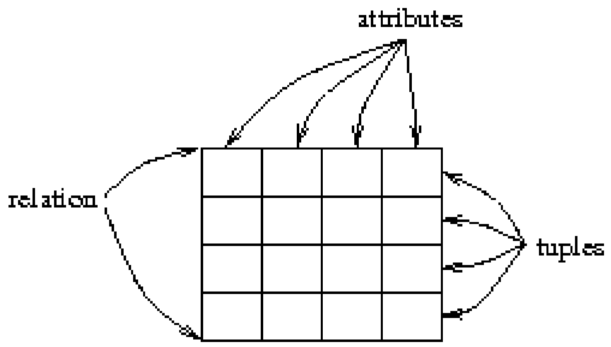


Figure 2. An overview of tuples and attributes in a relation

1. *Persistent storage*. When a user stores data on the disk, he or she wants the data to stay there, unless explicitly deleted. If the data is deleted, it is assumed that this is done purposely. Having a persistent storage, the user wants to manipulate, delete and observe the data in the database. The user may also want to verify that nothing has been altered unintentionally.

2. *One-time transient queries*. When the user wishes to collect data from the database, a *query* is posed and the system calculates and outputs the results. Hence, the DBMS has a *programming interface* that the user interacts with. The DBMS first compiles i.e. parse, optimise, and transform and then executes the query in order to obtain the correct data in an effective manner. Figure 3 below shows the three major steps involved in query compilation.

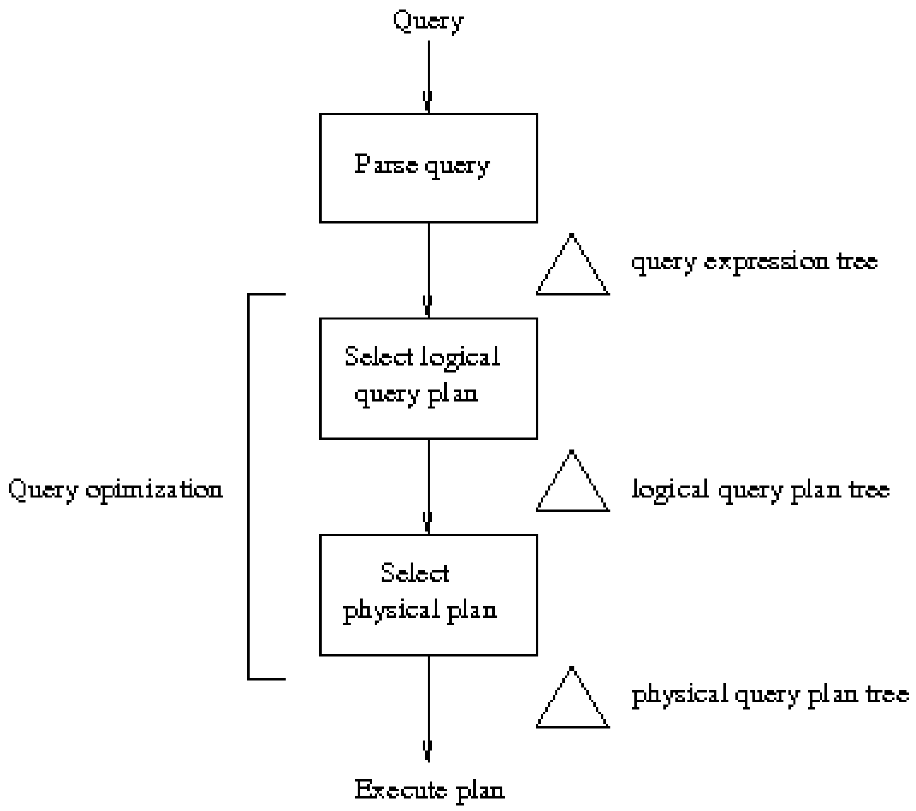


Figure 3. Outline of query compilation

The user has an idea of what he or she requires and writes the query to the computer. Usually, this query is expressed using a *query language* such as SQL, which is most commonly used (Garcia-Molina et al. 2002). The general structure of an SQL statement is given as follows:

```
SELECT <attributes>
FROM <realtions>
[WHERE <conditions>]
[GROUP BY <attributes>]
[HAVING <quality>]
```

The `SELECT` clause provides the projection of attributes and the `FROM` clause informs what relations to obtain the tuples from. The three following clauses are optional. Following the `WHERE` clause is a list of conditions that the tuples must satisfy in order to match the query. The `GROUP BY` clause divides the relations into groups based on the values in the list of attributes. The `HAVING` clause allows selection of certain groups in a way that depends on the group as whole, rather than the individual tuples. Subsequent to the query being registered, the parser converts it into a parse tree while checking the syntax of the query. At the following stage, the query semantics are checked and the query is converted into a logical query plan. This query plan may be optimised by the system after certain rules (e.g. analysing the size of the relations and number of attributes queried for) or by using heuristics about relations if there are no meta-data about the relations available. The system finally turns the logical query tree into the best possible physical plan, which is then executed.

3. *Random access*. Storage systems, where data may be stored and accessed in any order independent of the when it was originally recorded, are random access. Prior to being processed the data elements are read into main memory from the storage device. This is performed by specifying what block numbers the relations are stored in. If the relation uses *dense indexing* the tuples on the

storage device may be accessed directly. Since the data is stored persistently, knowing where the data is located will be sufficient for retrieval.

4. By claiming persistent storage, it is expected that the storage space has to be viewed as unbounded or infinite by the DBMS. This is because the system should be able to store new tuples without needing to delete previously stored data. Additionally, the idea is that the system should not have any pre-fabricated limit of storage, even though this is the case when using a device with limited storage capabilities. If a disk is full, a new disk is merely added.

5. DBMSs often provide a multi-user environment. This environment has to ensure that only one user at the time may have write access to a given element, such that inconsistent states cannot be found. In the case of system errors, the DBMSs may also need to re-create the last working state. This is managed by logging all critical operations, and is handled by the *transaction manager*.

6. *Only current state matters*. The DBMS does not use any historical to for instance optimising the queries. This is not consistent with the previous point where the transaction manager provided an overview of the transactions over time. As previously mentioned, a query is optimised after certain rules and heuristics that are based on e.g. knowledge about the size of the relations used within the query. When selecting the best possible physical query plan, the main optimising factor is the number of disk I/Os. Relative to accessing main memory, disk I/Os are extremely time consuming.

7. The traditional DBMS does not support any *real-time services* such as having deadlines determining *when* the result should be output. This denotes that the system does not throw away tuples if a certain time limit is exceeded due to a complex query or large amounts of data.

8. It is assumed that the DBMS always returns the *precise* answer to a query, because of the prior capability. This may, as stated above, affect the time consumption and hence reduce real-time support.

9. A DBMS has relatively low update rate, which is the rate for insertion of new tuples. Insertions, updates, and deletions usually occur less frequently than queries. Consequently, the DBMS focuses on storing the data in a manner that makes retrieval as efficient as possible. An example is to store a relation in continuous blocks on the hard disk. This may force the content within blocks to be reorganised as new tuples are stored, but reduces *seek time* i.e. the time the disk head uses to move from track to track. Another example is the different types of indexing. *Dense* indexing results in costlier insertions, however a cheaper lookups. Vice versa is recognised for *sparse* indexing.

As indicated above, there are several points that describe the DBMS. The following section discusses some techniques that assist the DBMS in fulfilling the requirement of precise results. These techniques assist in processing queries over relations that are too large to fit into main memory. In such large relations, a considerable amount of disk I/Os are needed to produce the correct result. These techniques are known as one-pass, two-pass, and multi-pass algorithms and are not applied on operators that process one tuple at the time such as selection and projection. An example of a query operator the above techniques may be applied on, is an equijoin between two relations *R* and *S* on an attribute *a*, which we denote $R \bowtie_a S$. We use the equijoins as an example even though there are several other operators that may require n-pass algorithms. Understanding the n-pass algorithms assists in the comprehension of the complexity that may occur for large amounts of data.

One-pass algorithms

One-pass algorithms are used when one of the relations fits into the memory. The available capacity of the memory is *M*. The relation *R* is size *B(R)* blocks, and

$$B(R) \leq M - 1.$$

This means that $B(R)$ is equal or smaller than the available memory minus one memory block. If the DBMS performs a join between *R* and *S*, *R* is read into the $M - 1$ blocks while *S* is read, block for block, into remaining memory block. In memory, the tuples from *S* are joined with corresponding tuples of *R*, based on equality of the attributes, and sent to output. Figure 4 below illustrates the one-pass algorithm. The lower block is the one applied to hold the block from *S*.

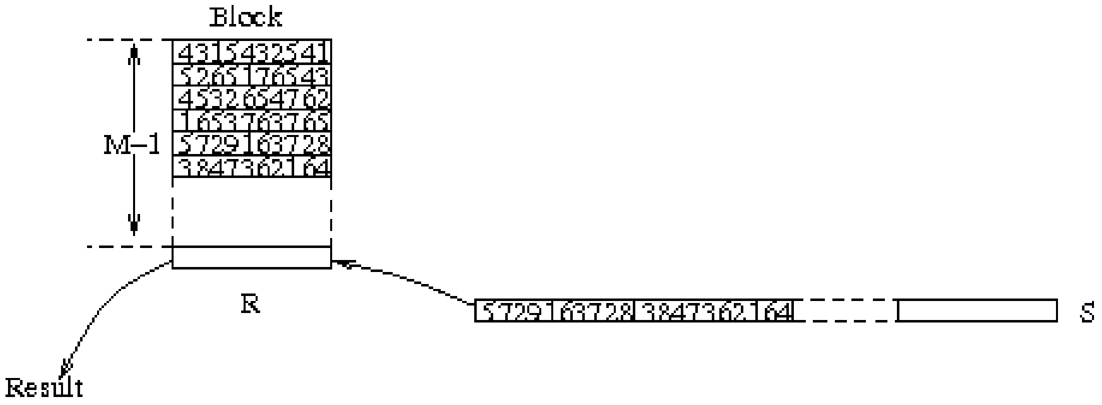


Figure 4. The one pass algorithm

Two-pass algorithms

The idea behind the two-pass algorithms is that it handles relations of size

$$M < B(R) \leq M^2.$$

In these cases, only parts of *R* are inserted into memory, *sorted* on some predefined criteria such as an attribute's value, and subsequently written back to the hard disk. This is considered the *first pass*. In the *second pass*, *M* sorted blocks are inserted into memory and processed at the time. Figure 5 below illustrates how the two-pass join algorithm works. As indicated by the figure, when joining, each of the relations are sorted according to the joining attributes, and inserted into memory before they are joined on e.g. equality. One solution is to *merge* the relations. A consequence is that if for example *S* contains more of a certain value than *R*, *R* waits until a new value appears in *S*.

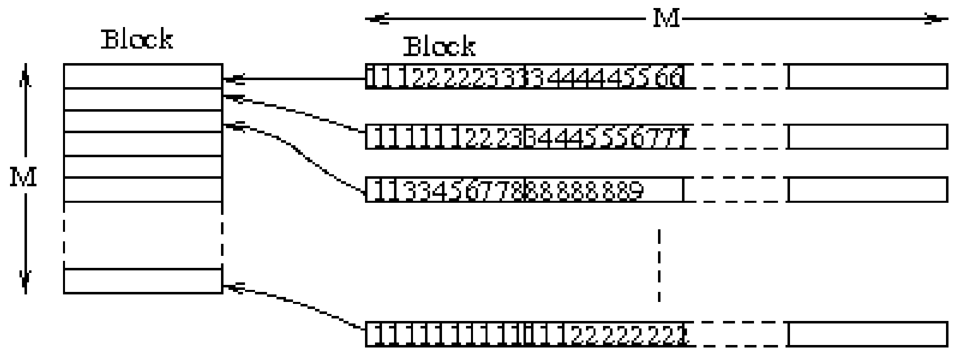


Figure 5. The two pass algorithm

Multi-pass algorithms

The multi-pass algorithms are employed when

$$M^2 < B(R).$$

These are more complex, hence forcing considerably more disk I/Os. As the two-pass algorithm sorts the relations into $O(M)$ one-pass relations, the multi-pass algorithm recursively sorts into $O(M^2)$ two-pass relations, which in turn sorts up to one-pass relations. These three algorithms reveal how the size of a relation affects the storage device and the amount of data it is forced to send and receive to obtain correct results.

3.3 Data Stream Management Systems

In contradiction to a DBMS that focuses on querying a database, the DSMS focuses on querying *streams of data*. Consequently, the DSMS's architecture is different compared to the list above. In the following, we compare the two systems' design to show the differences between them.

1. Instead of persistent relations, the DSMS aims to handle *transient streams*. In DSMSs, disk storage is not an issue, because data enters and leaves the system at possibly high rates. Data elements remain in main memory only for a limited amount of time. This suggests an architecture opposite to the DBMS's. However, some DSMSs integrate a DBMS to allow queries over both streams

and relations, providing the user with the possibility of for instance joining streams and relations. An example is TelegraphCQ (Chandrasekaran et al. 2003).

2. In contradiction to a DBMS, which executes the query one time, the DSMS uses *continuous queries*, which are queries that continuously produce results based on the tuples within the streams. In other words, the DBMS supports transient queries over persistent data, while the DSMS supports persistent queries over transient data. This issue is described thoroughly in Section 3.4.1. A DSMS may process several concurrent queries continuously over several streams. In a network monitoring scenario, several concurrent queries may aim to obtain different information from the network.

3. *Sequential access*. Since the data arrives as a stream, the DSMS reviews the tuples as a linear sequence, and does not have access to the data before or after the access interval. Sequential access is the opposite of random access, which is used in a DBMS. Some relational operators e.g. aggregations and most joins are blocking, which means that all input tuples to a query must be seen before any output may be produced. This is not possible over a stream of data, because data streams are considered infinite. When operating over streams, the DSMSs have to support *windowing* to unblock blocking operators. This means that calculations are performed on small partitions of the stream. These partitions are located in main memory, which implies limitations with regard to window size and accuracy. This will be discussed further in Section 3.4.1.

4. The DSMS does not generally aim to store tuples as they arrive in the system, because of the heavy volume and high rates of data streams. Expensive and time consuming disk I/Os cannot be allowed. Consequently, the DSMS is restricted by the size of the available memory. This means, as deducted in the description of the n-pass algorithms, that only *one-pass* algorithms may be used.

5. Due to optimising, the DBMS has a set of rules which are introduced in the query rewriting process, which is part of the query compilation as illustrated in Figure 3. An example of an optimising rule is to push projections as far as possible down towards the source i.e. to the database or the data stream. By doing this, less attributes are sent to the next operators, hence reducing the load. These rules also play a role in the DSMS, but the DSMS needs to adapt to the stream as well, by optimising the query tree on the fly, or re-allocating queue sizes.

6. A data stream may arrive at very high rates. Consequently, the DSMS must process the data in real-time i.e. as fast as the tuples arrive in the system. If the DSMS cannot keep pace with the data arrival rate, it may be forced to throw discard tuples that it does not manage to compute. Generally, this means that not all the tuples may be computed and that the DSMS has to support a set of *approximation algorithms*, which deliver results that for instance give a *sample* of the discarded tuples. Consequently, a DSMS possibly delivers approximately correct results in real-time, while a DBMS delivers precise results with the possibility of large delays. These large delays may occur if complex queries are posed over relations of several gigabytes.

Figure 6 below adapted from Golab et al. (2003), illustrates an abstract architecture model of a DSMS. One or more streams enter the system and are processed by the query processor. State information might be sent between the input monitor and the query processor to inform about e.g. stream characteristics, such that optimising and adaptations may be performed. When the query operators are finished processing the tuples, the result is sent to the output buffer.

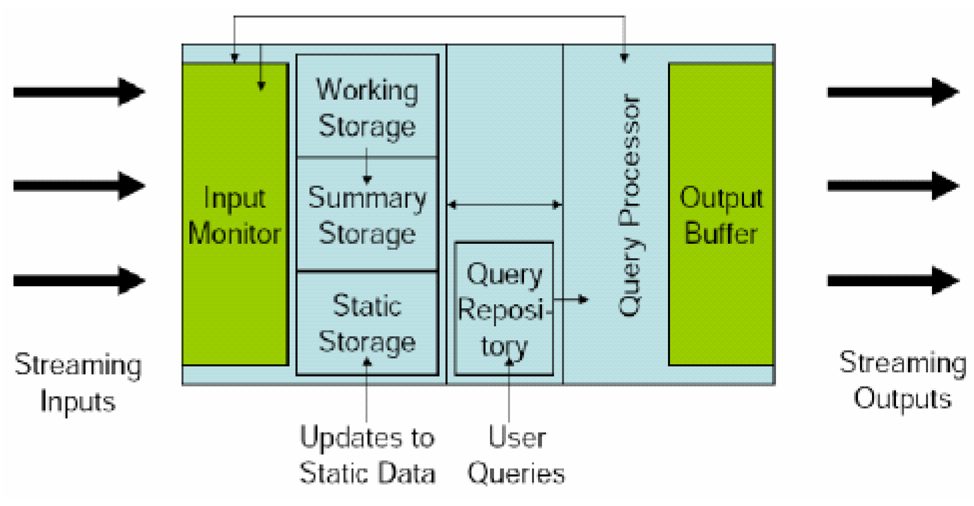


Figure 6. DSMS architecture

3.4 Issues in Data Stream Management Systems

The previous sections have presented an overview and introduction to DSMSs in addition to describing the main differences between DBMSs and DSMSs. In the following, we explain and discuss the most important issues in DSMSs. Initially, we will discuss some of the challenges that are associated with continuous queries and windowing in Section 3.4.1. In Section 3.4.2, we outline approximation and techniques for optimising the queries. A short overview of query language qualities and usages are given in Section 3.4.3. Finally, a short description of some of the existing DSMSs is presented in Section 3.4.4.

3.4.1 Continuous Queries and Time Windows

This section starts by showing how continuous queries have evolved from their first appearance in append-only databases to how they are employed in DSMSs at present. An append-only database is a database that only adds tuples, without deleting or updating them. Research regarding continuous queries allegedly started on such database systems (Golab et al. 2003). Terry et al. (1992) state that a user sometimes wishes to receive the tuples exactly when they are inserted into the database. Such tuples can have time critical values, and are only valid or interesting for a short period. Thus, there are requirements regarding a mechanism supporting such functionality. Based on these requirements, Terry et al. (1992) introduce the term *continuous queries* (*CQs*) i.e. queries that continuously search the database to reveal if any new entry or tuple has arrived, and possibly report the results if the tuples matches certain criteria.

3.4.2 The CQ's Building Blocks

CQs raise other issues than transient queries. To query the data stream, several suggestions on semantics have been proposed (Arasu et al. 2004b; Krämer et al. 2005; Terry et al. 1992). Throughout the thesis, we make use of the definition stated by Terry et al. (1992):

Definition 2 (Continuous semantics): The results of a continuous query are the set of data that would be returned if the query were executed at all instants in time.

Formally, this means that if $A(Q,\tau)$ is the set that is returned by running the query Q over a data set at time τ, $A(Q,\mathrm{T})$ denotes the union of all sets over a time interval $\tau \rightarrow \mathrm{T}$, then (Golab et al. 2003)

Equation 1: $A(Q,T) = \bigcup_{\tau=1}^{\mathrm{T}} (A(Q,\tau) - A(Q,\tau-1)) \cup A(Q,0)$.

Equation 1 shows that only the newest arriving tuples are handled by the query. This equation assumes that the database has a *monotonic* behaviour. *Monotonicity* is defined by a preservation of order. Thus, for a monotonic function f, iff $x \leq y$, then $f(x) \leq f(y)$. In the instance of a database, this may not always be the case. Sometimes tuples are altered after they are inserted. Therefore, a *non-monotonic* query may sometimes give a more correct result (Golab et al. 2003)

Equation 2: $A(Q,T) = \bigcup_{\tau=0}^{\mathrm{T}} A(Q,\tau)$.

The problem with the non-monotonic query is that it has to query the whole data set for each round. Given a CQ, this means that the whole set has to be queried continuously. With an increasing set i.e. tuples are added to the system without being deleted, the time consumed, whenever the query is run, will increase linearly.

The major challenge in using the semantics in Equation 1 is to preserve this monotonicity over a continuous stream in a deterministic way. By deterministic, we mean that when the query has started, it will always give the same results over the same data stream. Terry et al. (1992) illustrate this by using the following example:

A database contains messages that are inserted each timestamp t. We want to get all the messages that have not yet been replied to.

This example shows that without any more information, this query may force the system to non-monotonic computing. When do we know that we have all the answers? Without any precision, this query returns all the messages as they arrive; because a message cannot be replied to before it is sent. If the database is append-only, we eventually get the correct results as more messages arrive, if the query runs indefinitely. The system needs to view all the previous messages to see if the new message is a reply to an older one. The problem is that an incoming data stream, as defined in Definition 1, may require a vast amount of storage. Hence, the database has to drop tuples to make the querying possible. This means that there is a risk of deleting messages before the replies actually appear.

Such an inconsistency leads to the term of *windowing* over a stream of data. A window is a partition over the stream which guarantees determinism inside it. This means that, given a plain limit, the example stated above may be rephrased to the following:

Get all the messages that have not been replied to within five minutes after it has been sent.

The query looks at all the messages and stores them. It also reduces the monotonicity by only re-calculating over a five minutes window, which in turn returns the correct tuples. Yet, re-calculations must occur, something that leads to *blocking* of the system over the time window.

Blocking may be critical for the DSMS. A blocking operator is a query operator that is unable to produce the first of its output until it has seen its entire input (Babcock et al. 2002). For example, the DBMS blocks while it performs two- or multi-pass operations; the data is obtained from the database e.g. sorted, and then written back to disk possibly several times before the final result is returned. This is viable when there is a finite bag of data being handled. As pointed out in the stream definition, a data stream is an *infinite* bag, thus reducing algorithms only in order to support one-pass.

In the context of joining, and that we still include the database in the DSMS model, it is possible to join between streams, and to join between streams and relations. For values between 1 and *n* we denote $S_1 \dots S_n$ for *n* streams and $R_1 \dots R_n$ for *n* relations. We set *i* and *j* such that $1 \le i \le n$ and $1 \le j \le n$. In case of $S_i \bowtie S_j$, it is required that $B(S_i) + B(S_j) \le M$. The latter statement explicitly reduces the window sizes so that the two windows together cannot contain more tuples than the size of the allocated memory. This is also the case for $R_i \bowtie S_j$. It has to be so that $B(R_i) \le M - B(S_j)$ to verify that the join is deterministic; if some of the tuples in R_i were temporarily written to the hard disk when the window of S_j was in memory, the query would have produced a wrong answer since some of the tuples were not available. Note that a stream and a relation have different qualities, and that we require correct results from the relation, according to the preceding DBMS presentation.

The following presents a discussion of some of the additional requirements that we outlined in Section 2.3.

The DSMS literature discusses three main window designs (Golab et al. 2003); the *sliding window*, the *jumping window*, and the *hopping* or *tumbling window*. In addition, Chandrasekaran et al. (2003) discuss the *landmark* window, which relates the prior discussion to the following. We start by giving a short description of this type of windows.

Landmark Window

The landmark windowing technique and the append-only database have much in common. Gehrke et al. (Gehrke et al. 2001) defines the landmark window to have a fixed point from which the window moves. The window increases in size while tuples are added. This solution poses many of the same challenges discussed above. One issue with the landmark windows is that they do not access historical data (Chandrasekaran et al. 2003). As we see in the following presentation, this is a common quality for all the windowing techniques, and thus a necessity to avoid blocking. The landmark window is illustrated in Figure 7(a) below. It shows that tuples are added as filled squares as time elapses. Time is indicated by the vertical arrow pointing downwards. The horizontal arrow shows how the window evolves. As perceived, the landmark window does not remove any tuples, thus the number of tuples in the window increases.

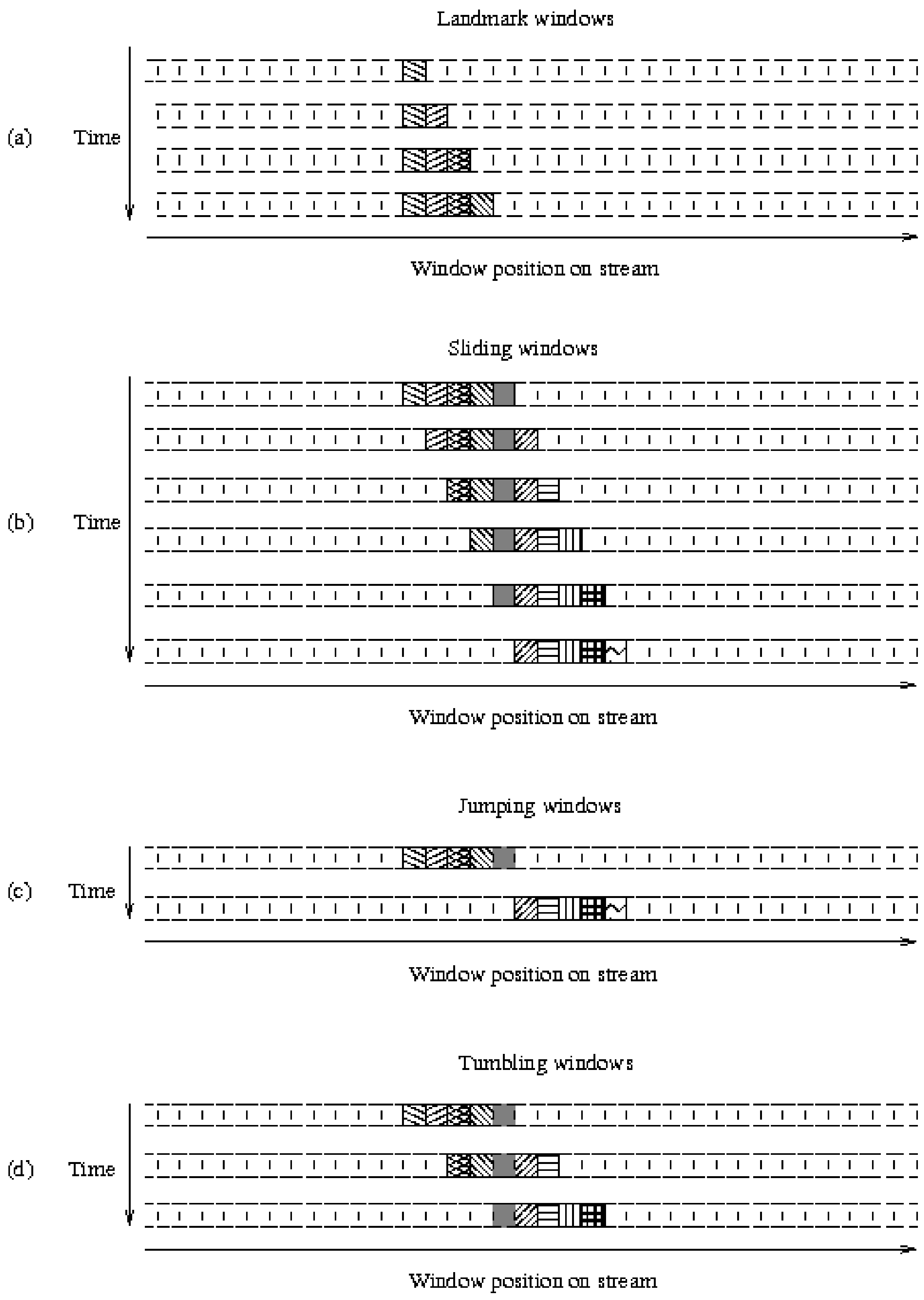

Landmark windows
(a)
Time
Window position on stream
Sliding windows
(b)
Time
Window position on stream
Jumping windows
(c)
Time
Window position on stream
Tumbling windows
(d)
Time
Window position on stream

Figure 7. Sliding, jumping, and tumbling windows

Sliding Window

The sliding window resembles the windowing technique we have discussed so far. A window has e.g. a specified time length, *l* and a specified time τ, so that $0 \leq l \leq \tau$. The example extracted from Terry et al. (1992) reveal a window of five minutes, and all messages that have not been replied to, will be deleted after this period. Strictly speaking, this means that a tuple that enters within the window borders remains inside the window for a period of *l* time units before it is overwritten or de-allocated.

Note that there are two different types of windows. *First, physical* windows make use of time as constraint i.e. they are specified by e.g. five minutes or ten seconds. Second, *logical* windows do not employ time as constraint, but rather use number of tuples to determine the window size. In regards to the latter, the required memory is known *a priory*, and the space may be allocated at the registration of a query. The physical windows need to allocate memory dynamically, since the stream's behaviour depends on fluctuations in the data arrival rate (Chandrasekaran et al. 2003).

Figure 7(b) above illustrates the sliding window. The sliding window is recalculated for each time stamp. One consequence is that, for the window length *l*, a tuple that matches a condition is reported *l* times, given that the recalculations are performed each time stamp. This is correct behaviour due to the specification, but not always what is intended.

Jumping Window

In cases where the recalculations in the sliding window are not required, the *jumping* window offers an alternative solution. Instead of *sliding* over the stream, the jumping window fills the window with tuples and performs the calculations. When the calculations are completed, it starts to fill up a new window. This removes the recalculation of the tuples, but does not ensure correct results, since for instance a message can acquire its reply in the next window. Figure 7(c) above illustrates the jumping window. The two sets of tuples are disjoint.

Tumbling Window

The third alternative is the *hopping* or *tumbling* window, which is illustrated in Figure 7(d) above. If the sliding window, w_s, is recalculated each time stamp, and a jumping window, w_j, is recalculated each *l*th time stamp, the tumbling window, w_t, can recalculate by a time interval T such that $\mathrm{T}(w_s) < \mathrm{T}(w_t) < \mathrm{T}(w_j)$.

In addition, a fourth alternative represents the cases where the update interval is larger than the window size. This causes pausing in the updates equivalent to the difference between the end of the update interval and the end of the window.

As mentioned, windows are utilised to unblock blocking operators like joins. When tuples from two windows intend to join, one of the windows scan the other while blocking the input streams.

3.4.3 Approximation and Optimisation

Approximation

Next to choosing the best window size semantics, the choice of window size depends on storage capacities, speed of the system, and input rate of the data stream. If, as a worst-case scenario, the average data rate is *M* i.e. the size of the memory, per time-stamp τ, the maximum window size T has to be set to $\mathrm{T} = \tau$. Such scenarios may lead to scarce system resources, since the available memory is only used for storing tuples. This must also be taken into consideration when using several concurrent queries. If each query uses a window of size *M,* the system runs out of resources as soon as the queries require the DSMS to allocate memory.

The main solution is to reduce the number of tuples. In such cases, the DSMS has to maintain some mechanisms for removing, however, still registering the discarded tuples known as *shedded* tuples. The following list sums up several of the different solutions suggested in the DSMS literature (Golab et al. 2003):

- *Counting.* The method typically stores frequency counts for attributes selected in a query. This may be useful for applications where the frequency of items is important, but where the tuples' details are not in focus.

- *Hashing* uses functions to hash the dropped tuples into *n* buckets, such that the frequency is incremented per match in the hash table. This may be helpful if the stream contains several equal tuples.

- *Sampling* reports small random samples of the streams. These random samples often represents the stream well. However, some queries (e.g. finding the sum of a stream) may not be reliably computed by sampling.

- *Sketches* use a random number chosen from some distribution with a known expectation to decide for the tuples' appearance. All other tuples are shedded.

- *Wavelets* are based on the same idea as sketches; the choosing of tuples is based on calculations on probability of appearance over a large set of tuples.

Optimisation and Adaption

The memory consumption is a relatively important factor when handling the data streams. Queries that only filter tuples from a single stream do not require any extensive amount of memory; each tuple is compared to the filter and either ignored or sent to output. With regards to queries that use aggregating operators like average, or operators like join, some optimisation may be required. As mentioned earlier in this section, a general optimising technique is to push projections as far as possible to the source (Garcia-Molina et al. 2002) i.e. the entering data stream. For example, given the two relational data streams $S_1(A, B, C)$ and $S_2(C, D, E)$, and the query

$$\pi_A(S_1 \bowtie S_2).$$

This query may be optimised such that

$$(\pi_{A,C}(S_1)) \bowtie (\pi_C(S_2)).$$

However, there are restrictions to pushing projections down the query tree. For example, if an operator further up the query tree projects other attributes than the current operator, it is important that the current operator projects these tuples as well.

When joining two streams that arrive in different rates, it is also important to consider reordering the joins i.e. adapting to the stream characteristics as they change. An example is to prefer that one specific window searches another for join equalities instead of the opposite. As an alternative, the Telegraph project has proposed a solution called an *eddy* (Avnur et al. 2000), which instead of sending tuples up a query tree, sends them to query operators which are connected only to the eddy.

In sensor networks, as described in Section 2, the sensors may perform local computation (e.g. filtering and aggregation) before they send their data towards the central node. In network monitoring, a router may collect data and perform local computation before sending the data to a final machine that performs the final calculations. Golab et al. (2003) suggest that it may be an optimising factor to distribute the query operators among the participating nodes or routers. The distribution of queries throughout a network may resemble optimisation of the query tree.

3.4.4 Query Languages

This far, the DSMS and its functionality have been formally described. However, there are several implementations of DSMSs. Those implementations use languages that are mainly inspired by the already existing DBMS languages. As stated in the DBMS overview, the most common query language over relational databases is SQL (Eisenberg et al. 1999). We base our discussion on SQL's declarative semantics, since SQL is the language that is most commonly utilised in relational modelling of data.

Note that one important aspect with DSMSs is that they use a declarative language, which makes it easy for the user to understand what the query is supposed to do. This makes the code more portable, and makes it easier to verify. Today many applications are written in Perl or other high-level languages, which often makes it complicated for other users to understand the source code, as it tends to be complex, and not well documented. The declarative languages therefore make it easier to share knowledge. They also create a common platform for discussing, understanding, and further develop the actual query (Plagemann et al. 2004).

If a query includes blocking operators such as aggregations, it is necessary to add window semantics to unblock the query. If the current DSMS supports the possibility of joining several streams, it is necessary to specify one window for each stream. In addition, the type of window must be specified if the DSMS provides support for several windowing techniques. Following, is a general example of windows.

In the early versions of the TelegraphCQ DSMS, a for-loop construct with a variable `t` that iterates over time followed each query definition. The type and size of the window is specified by a `WindowIs()` statement in the loop. Let `S` be a stream and `ST` the start time of a query. To specify a sliding window over `S` with size five that should run for fifty days, the following for-loop may be appended to the query:

```
for (t = ST; t < ST + 50; t++)
   WindowIs(S, t - 4, t)
```

By changing the variables in the above expression, it is possible to support all the four windowing techniques described in this section. For example, a jumping window may be expressed by changing the increment condition in the for-loop to `t = t + 5`.

Other examples of CQ models given by Golab et al. (2003) are the list-based models, the time series models, and the sequence models. Either the query languages of these models add extensions to SQL, which makes them better suited for their applications, or they introduce alternative languages, such as procedural languages, where the user defines the streams and operators by drawing arrows and boxes, respectively. However, this is beyond the scope of this thesis.

As in SQL99, the possibility of using sub-queries may also increase the expressiveness of queries in continuous query languages. In theory, it is possible to support sub-queries if the DSMS supports multiple queries. However, we have experienced that there are still limitations in how sub-queries may be expressed in some of the current CQLs.

3.4.5 Examples of DSMSs

Since the data stream systems have been a topic of excitement the last few years (Golab et al. 2003) and several have been developed, the current section is summarised by presenting a short overview of some of the DSMSs that exist today. Most of the information is adapted from Golab et al. (2003) and Goebel et al. (Goebel et al. 2005). We investigate the DSMSs with regard to the applications for which they are intended, the input they accept, the operators they use, the windowing technique, languages, and if possible, optimisation and adaptation.

Aurora and Medusa

Aurora (Abadi et al. 2003) is developed at Brown University and Brandeis University and mainly focus on querying sensor data. In Aurora, both streams and *tables* are used as data input. Abadi et al. (2003) use the term *static table* to describe windows on streams with unlimited size. Aurora has a query algebra called SQuAl (*S*tream *Q*uery *Al*gebra), which uses a procedural language as described in the prior sections. An operator manipulates boxes, as illustrated in Figure 8 below. The boxes are represented by a set of operators used for e.g. filtering, sorting, mapping, aggregating, union and joining. According to Golab et al. (2003), Aurora also has support for fixed, landmark, and sliding windows. Aurora uses several optimisation techniques e.g. inserting projections, and combining and re-ordering boxes. During run-time, a QoS monitor investigates the streams and discovers whether optimisation is required. When Aurora is used in tight couplings between several machines, it is called Aurora*.

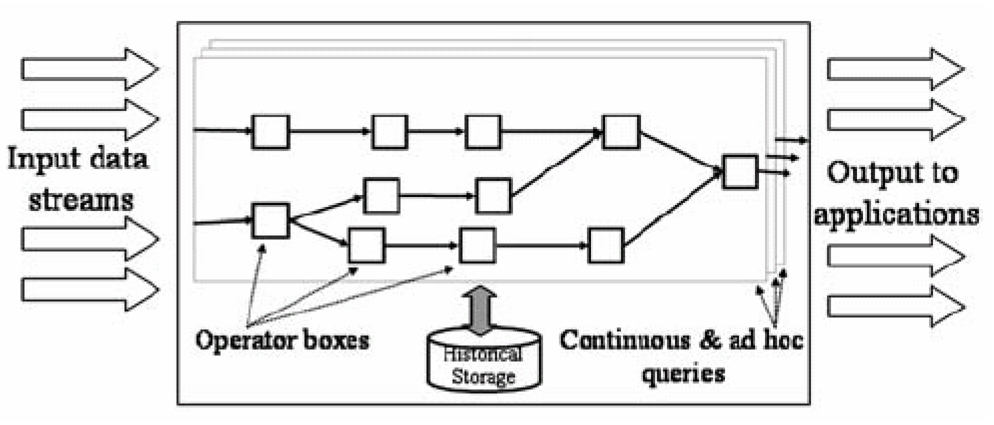


Figure 8. An overview of the Aurora DSMS

As Aurora is the few-node query engine, Medusa (Zdonik et al. 2003) offers a solution for distributing Aurora over multiple nodes and organisation networks. Medusa is developed at MIT (Massachusetts Institute of Technology), and focuses on the inter-network challenges like efficient TCP/IP multiplexing of several connections.

Borealis

Borealis (Balazinska et al. 2005) inherits elements from both Aurora and Medusa. The system is developed as a collaboration between MIT, Brown University, and Brandeis University. It handles both inter node query processing and the distribution of several nodes over large networks. The idea of distributing the system is that it has an incremental scalability in case of e.g. high load spikes, and high availability, to monitor the system's health and perform fast fail-over. These parallel processing issues help Borealis act dynamically.

Gigascope

The Gigascope (Cranor et al. 2002) DSMS is developed at AT&T (American Telephone and Telegraph Company) to monitor network traffic at ISPs. The system is distributed in a way that some query operators are pushed to the routers to collect interesting information. Gigascope uses a query language called GSQL, which supports selection, join, aggregation, and *stream merge*. In contrast to the relation-based model, Gigascope operates on the data streams directly instead of transforming the data. Gigascope also tries to avoid blocking operators by assuming monotonicity and ordering property on the joining attributes. If, for example, two joining streams, R and S, have increasing sequence numbers a, one can join $R \underset{R.a < S.a}{\bowtie} S$ by simply monitoring the a values as they arrive. Since Gigascope's intention is to obtain data from simplex fibre optic lines, one requires data from several interfaces to gain an overview of the stream. Thus, the stream *merge* operator is used to perform a union on the two streams before e.g. joining them. This may be considered an optimisation technique, besides from rearranging operators, which optimises Gigascope as well. If the two streams have differing rates, there might be an overflow in the merge buffers. This is solved by inserting punctuation tuples in the stream to release the waiting stream. Both Aurora and Gigascope are proprietary DSMSs that aim to work commercially in AT&T's networks.

Niagara

As both Aurora and Gigascope focus on low level data streams, Niagara (Chen et al. 2000) supports data streams from Web-pages and Web searching. The DSMS aims to join millions of continuous queries by grouping similar queries together because several queries may share equalities. To identify similarities, Niagara's continuous query language (NiagaraCQ) uses an XML-like syntax to execute the multiple continuous queries. The queries are e.g. expressed as follows (Chen et al. 2000):

```
Where <Quotes> <Quote>
   <Symbol>INTC</>
   </> </> element_as $g
in ''http://www.cs.wisc.edu/db/quotes.xml''
construct $g
```

The query obtains tuples from `quotes.xml` and projects the values from the field `INTC`. If the equality operator is used and many queries gain results from the same source (e.g. a stock exchange XML file) an XML table is constructed to include the equalities and destinations in the source files.

STREAM

STREAM is developed at the Stanford University, and is a general purpose DSMS that aims to investigate resource sharing and adaptive query processing. The input stream is tuples, and the attribute data types can be integer, char(*n*), float, or byte. STREAM supports the sliding windowing technique to unblock the data streams. In addition, it uses a set of operators; *stream-to-relation*, *relation-to-relation*, and *relation-to-stream*. The stream-to-relation operator employs the windowing technique to map the data stream to a relation. The relation-to-relation operator uses SQL to operate on the tuples, which are represented in a relation. Finally, the relation-to-stream operator may stream the content of the relation in three different ways, as described in Section 4.

TelegraphCQ

TelegraphCQ (Chandrasekaran et al. 2003) is developed at UC Berkeley and aims to be a general purpose relational DSMS. It is constructed as part of the public domain DBMS PostgreSQL and inherits much of PostgreSQL's functionality. Nevertheless, TelegraphCQ offers some adaptation techniques distinguishing it from PostgreSQL. This also implies that TelegraphCQ is not implemented including all the functions PostgreSQL supports. The query syntax used in TelegraphCQ is called *StreaQuel*, and is similar to SQL except from the windowing semantics. A module called an *eddy* creates the main adaptivity. This module sends and receives tuples that are processed by different operators. This poses an alternative to the static query tree that is utilised by other DSMSs. TelegraphCQ's windowing technique makes it possible to use sliding, jumping, and tumbling windows. In contradiction to STREAM, TelegraphCQ only manages to use one type of streams, which is created using a `CREATE STREAM` statement. However, TelegraphCQ provides functionality for storing the stream to disk such that it later can be queried as a relation in PostgreSQL.

4. STREAM

In the STREAM project at Stanford University, California, they are investigating data management and query processing for the class of applications that we discussed in Section 2. As part of their project, they are building a general-purpose prototype DSMS which supports a large class of declarative continuous queries over continuous streams and traditional stored data sets. This prototype is also known as STREAM. The STREAM prototype targets environments where streams may be rapid, stream characteristics and query loads may vary over time, and system resources may be limited. The first STREAM prototype, STREAM 0.5.0, was released as a public domain DSMS in November 2004. In February 2005, the second prototype, STREAM 0.6.0, was issued. Even though we are familiar with the first prototype, we base all discussions and analyses in the current thesis on STREAM 0.6.0. When there is no ambiguity present, we use the term "STREAM" interchangeably to describe both the project and the name of the prototype.

4.1 The Continuous Query Language (CQL)

For simple continuous queries over streams, it may be sufficient to use a relational query language (e.g. SQL) replacing references to relations with references to streams, registering the query with the stream processor, and waiting for answers to arrive (Arasu et al. 2003b). For simple monotonic queries over complete stream histories, this approach is nearly sufficient. However, as queries grow more complex (e.g. with the addition of aggregation, sub-queries, windowing constructs, and joins of streams and relations) the semantics of a conventional relational language applied to these queries quickly becomes unclear (Arasu et al. 2004a). In the STREAM project, they have defined a formal *abstract semantics* to address this problem. In addition, they have designed CQL, which implements the abstract semantics. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and updatable relations.

4.1.1 Streams and Relations

The abstract semantics are based on two data types, *streams* and *relations*. In this section, we show how a formal model of streams and updatable relations are defined. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of named attributes. A discrete, ordered time domain T is introduced for stream element arrivals and relation updates. A *time instant* is any value from T. Time domain T models an application's notion of time, not particularly system or wall-clock time. As we mention in Section 3.1, we use one definition of a stream throughout the thesis. The definition is repeated below for convenience:

(**Stream**) A stream *S* is a possibly infinite bag (multi-set) of *elements* $\langle s, \tau \rangle$, where *s* is a tuple belonging to the schema of *S* and $\tau \in \mathrm{T}$ is the *timestamp* of the element. There is a finite but unbounded number of stream elements for any given timestamp $\tau \in \mathrm{T}$.

There are two classes of streams: *base streams*, which are the source data streams that arrive at the DSMS, and *derived streams*, which are intermediate streams produced by operators in a query.

Definition 3 (**Relation**) A relation *R* is a mapping from T to a finite but unbounded bag of tuples belonging to the schema of *R*.

A relation *R* defines an unordered bag of tuples at any time instant $\tau \in \mathrm{T}$, denoted $R(\tau)$. Note the difference between this definition for a relation and the standard one: in the standard relational model, a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned.

4.1.2 Abstract Semantics

In addition to being based on the two data types, streams and relations, the abstract semantics is based on three classes of operators over streams and relations. These classes are shown in Figure 9 below adapted from Arasu et al. (2004a).

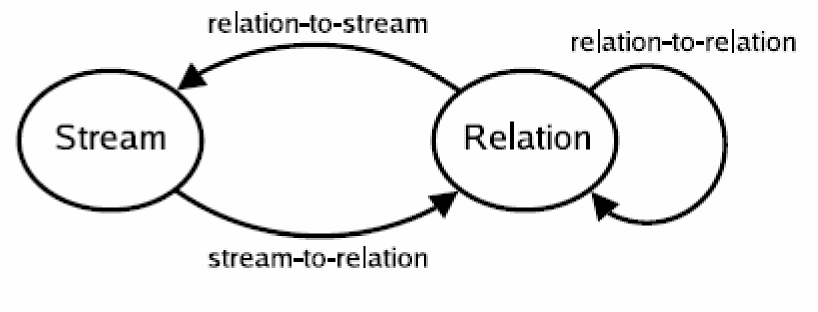


Figure 9. Classes of operators in abstract semantics

- A *stream-to-relation* operator takes a stream as input and produces a relation as output.
- A *relation-to-relation* operator takes one or more relations as input and produces a relation as output.
- A *relation-to-stream* operator takes a relation as input and produces a stream as output.

Stream-to-stream operators are absent; they are composed from operators of the above three classes. The inputs to a continuous query are either streams or relations, and the output is either a stream or a relation, depending on the class of the root operator in the tree representing the query. Arasu et al. (2003b) define the abstract semantics presented by the STREAM project:

Definition 4 (**Continuous Semantics**) Consider a query Q that is any type-consistent composition of operators from the above three classes. Suppose the set of all inputs to the innermost (leaf) operators of Q are streams $S_1,...,S_n$ $(n \geq 0)$ and relations

$R_1,...,R_m$ $(m \geq 0)$. We define the result of continuous query *Q at a time* τ, which denotes the result of *Q* once all inputs up to τ are "available." There are two cases:

- Case 1: The outermost (topmost) operator in *Q* is relation-to-stream, producing a stream *S* (say). The result of *Q* at time τ is *S* up to τ, produced by recursively applying the operators comprising *Q* to streams $S_1,...,S_n$ up to τ and relations $R_1,...,R_m$ up to τ.

- Case 2: The outermost (topmost) operator in *Q* is stream-to-relation or relation-to-relation, producing a relation *R* (say). The result of *Q* at time τ is $R(\tau)$, produced by recursively applying the operators comprising *Q* to streams $S_1,...,S_n$ up to τ and relations $R_1,...,R_m$ up to τ.

4.1.3 Continuous Query Language

In this section, we present the operators that constitute the concrete CQL language. These operators are defined by implementing the above abstract semantics.

Stream-to-Relation Operators

Currently, all stream-to-relation operators in CQL are based on the concept of a *sliding window* over a stream; a window that at any point of time contains a historical snapshot of a finite portion of the stream (Arasu et al. 2003b). There are three classes of sliding window operators in CQL: *time-based*, *tuple-based*, and *partitioned*. These windows are expressed using a window specification language derived from SQL-99 (Arasu et al. 2004a).

Time-Based Sliding Window

A time-based sliding window on a stream *S* takes a time interval T as a parameter and produces a relation *R*. It is specified by following the reference to *S* with [Range T]. More formally, the output relation *R* of "S [Range T]" is defined as (Arasu et al. 2003b):

$$R(\tau) = \{s \mid \langle s, \tau' \rangle \in S \wedge (\tau' \leq \tau) \wedge (\tau' \geq \max\{\tau - \mathrm{T}, 0\})\}$$

Tuple-Based Sliding Window

A tuple-based sliding window on a stream *S* takes a positive integer *N* as a parameter and produces a relation *R*. It is specified by following the reference to *S* with [Rows *N*]. More formally, the output relation *R* of "S [Rows N]", $R(\tau)$ consists of the *N* tuples of *S* with the largest timestamps $\leq \tau$ (or all tuples if the length of *S* up to τ is $\leq N$) (Arasu et al. 2003b). However, when timestamps are not unique, tuple-based sliding windows may be nondeterministic. For example, given a sliding window of *N* tuples, several tuples with the *N*th most recent timestamp, and that the other *N* – 1 more recent timestamps are unique. Then one of the tuples with the *N*th most recent timestamp must be chosen in some fashion to generate exactly *N* tuples in the window.

Partitioned Sliding Windows

A partitioned sliding window on a stream *S* takes an integer *N* and a set of attributes $\{A_1,...,A_k\}$ of *S* as parameters and is specified by following *S* with [Partition By $A_1,...,A_k$ Rows N]. It logically partitions *S* into different sub-streams based on equality of attributes $A_1,...,A_k$ computes a tuple-based sliding window of size *N* independently on each sub-stream, then takes the union of these windows to produce the output relation. More formally, a tuple *s* with values $a_1,...,a_k$ for attributes $A_1,...,A_k$ occurs in output instantaneous relation $R(\tau)$ iff there exists an element $\langle s, \tau' \rangle \in S,\ \ \tau' \leq \tau$ such that τ is among the *N* largest timestamps of elements whose tuples have values $a_1,...,a_k$ for attributes $A_1,...,A_k$ (Arasu et al. 2003b).

Relation-to-Relation Operators

CQL uses SQL constructs to express its relation-to-relation operators, and much of the data manipulation in a typical CQL query is performed using these constructs. The current CQL implementation in STREAM offers only a small subset of the SQL operators that usually are provided through a DBMS.

Relation-to-Stream Operators

CQL has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*. In the following formal definitions, the operators are assumed to be the bag versions (Arasu et al. 2003b).

- `Istream` (for "insert stream") applied to relation R contains a stream element $<s, \tau>$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Assuming $R(-1) = \phi$ for notational simplicity, we have:

$$\texttt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

- Analogously, `Dstream` (for "delete stream") applied to relation R contains a stream element $<s, \tau>$ whenever tuple s is in $R(\tau - 1) - R(\tau)$. Formally:

$$\texttt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

- `Rstream` (for "relation stream") applied to relation R contains a stream element $<s, \tau>$ whenever tuple s is in $R(\tau)$. Formally:

$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

4.1.4 CQL Syntax

We illustrate the CQL syntax by giving some grammar rules. These rules only represent a subset of the CQL grammar. Note that the `HAVING` clause is not included in this illustration. The reason for this is that the `HAVING` clause is not supported in the current STREAM prototype. When a CQL query is parsed, it is converted into a *parse tree*, which is a tree whose nodes correspond to either *atoms* or *syntactic categories*. The representation of atoms and syntactic categories in the grammar are collected from (Garcia-Molina et al. 2002). "Non_mt" means "non empty," and "opt" means "optional." The symbol "`::=`" means "can be expressed as."

```
<Query>   ::=  SELECT <Opt_distinct> <Non_mt_select_clause>

               FROM <Non_mt_relation_list>

               <Opt_where_clause>

               <Opt_group_by_clause>

<Query>   ::=  <xstream_clause> ( <Query> )
```

All syntactic categories need further rules attached to them in order to define a complete grammar. One example that parses a syntactic category into atoms is (the symbol "|" means "or"):

```
<xstream_clause>    ::=  ISTREAM | DSTREAM | RSTREAM
```

4.1.5 Examples of CQL Queries

The following examples show how queries are expressed in CQL.

Example 1
The following continuous query filters a stream S:

```
ISTREAM(
SELECT * FROM S [ROWS 100]
WHERE S.A < 50)
```

Stream S is converted into a relation by applying a tuple-based sliding window. The relation-to-relation filter "S.A < 50" acts over this relation, and the inserts to the filtered relation are streamed as the result.

Example 2
The following query is a *windowed join* of two streams S1 and S2:

```
SELECT * FROM S1 [ROWS 100], S2 [RANGE 1 MINUTE]
WHERE S1.A = S2.A AND S1.A < 50
```

The answer to this query is a relation. At any given time, the answer relation contains the join (on attribute A with A < 50) of the last 100 tuples of S1 with the tuples of S2 that have arrived in the previous minute.

4.2 An Architectural Overview of STREAM

In this section, we present an architectural overview of the STREAM system. In addition, we introduce some important concepts and show how they are linked together.

4.2.1 High-Level System Architecture

A simplified and high-level view of STREAM is shown in Figure 10 below adapted from Arasu et al. (2003a). On the left hand side are the incoming data streams, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires an intermediate state, which is labelled *Scratch Store* in the figure. Although the main concern is online processing of continuous queries, in many applications stream data may also be copied to an *Archive* for preservation and possible offline processing of expensive analysis or mining queries (Arasu et al. 2003a). At the top of the figure, we see users or applications registering continuous queries, which remain active within the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time (similar to materialised views).

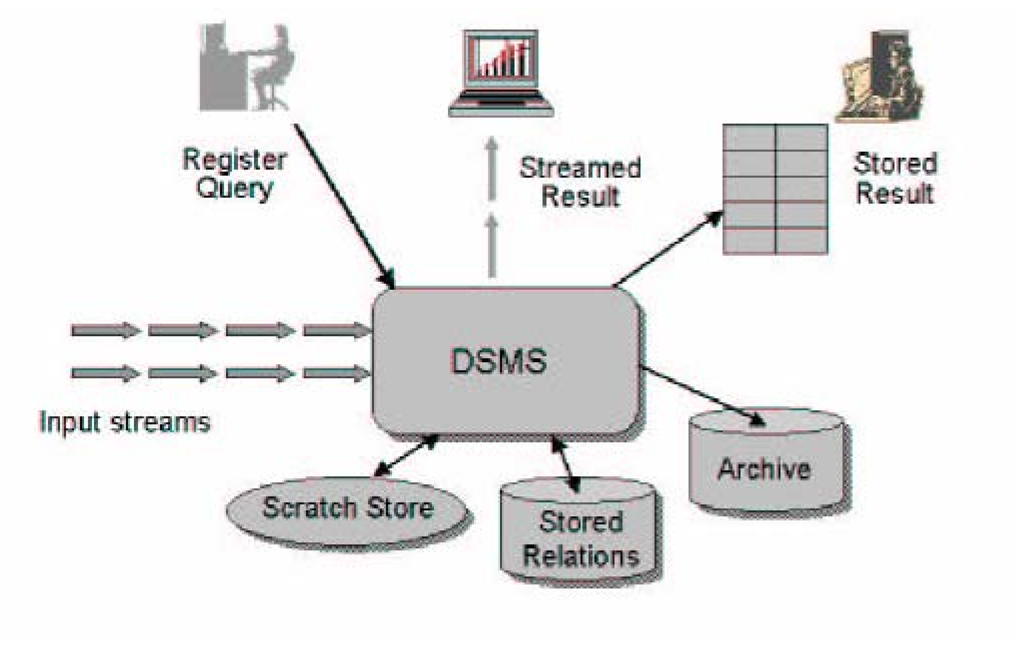


Figure 10. Overview of STREAM

4.2.2 The STREAM System Interface

In the STREAM project, they have developed a graphical *query and system visualiser* for the STREAM system. The visualiser allows the user to (Arasu et al. 2004a):

- View the structure of query plans and their component *entities* (operators, queues, and synopses).
- View the detailed properties of each entity.
- Dynamically adjust entity properties.
- View *monitoring graphs* that display time-varying entity properties such as queue sizes, throughput, overall memory usage, and join selectivity, draw dynamically against time.

The visualiser is an extension to STREAM and takes the form of a client, transferring tuples to the server (DSMS) via TCP connections. Results are streamed back to the client. In our analysis, we only consider the server side of this composition. We use a generic command-line client, which was released together with the prototype. This client implements the external interface of STREAM, nevertheless, the implementation of the client only streams data from files and into the system. Consequently, we change the client to make it possible to stream live network data into STREAM. In Section 6.1, we will describe how we make these changes. We do not discuss the visualiser any further, because it does not take any part of our analysis. The generic command-line client will be described in Section 4.4.1.

4.2.3 Query Plan

When a continuous query specified in CQL is registered with the STREAM system, a *query plan* is compiled from it. Query plans are trees consisting of *operators*, *queues*, and *synopses*. Operators perform the actual processing, and they are connected by queues, which buffers tuples (or references to tuples) as they move between operators. Synopses, which store operator data, are connected to operators as required (Arasu et al. 2004a). The query plan is merged with existing query plans whenever possible in order to share computation and state (Arasu et al. 2003b).

Example Query Plan

We use the query in Example 2, Section 4.1.5, in order to illustrate how a query plan is generated. The original query is repeated here for convenience:

```
SELECT * FROM S1 [ROWS 100], S2 [RANGE 1 MINUTE]
WHERE S1.A = S2.A AND S1.A < 50
```

The query plan corresponding to this query is shown in Figure 11 below adapted from Arasu et al. (2004a).

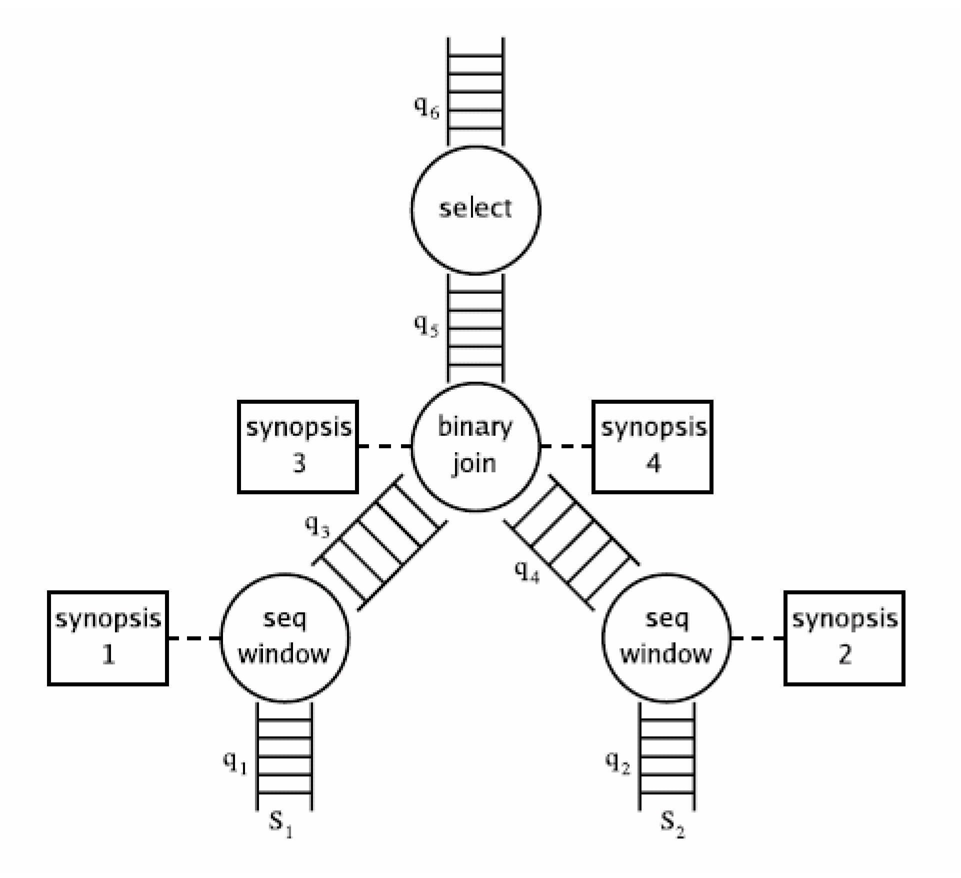


Figure 11. A query plan illustrating operators, queues, and synopses

A careful reader may observe that the query plan may be optimised by pushing the `select` operator down into one or both branches below the `binary-join` operator, and also below the `seq-window` operator on S2. However, Arasu et al. (2004a) states that tuple-based windows do not commute with filter conditions, and therefore the `select` operator cannot be pushed below the `seq-window` operator on S1.

Note that the contents of *synopsis1* and *synopsis3* are similar (as are the contents of *synopsis2* and *synopsis4*), since both maintain a materialisation of the same window, but at slightly different positions in stream S1. In Section 4.3.3, we will demonstrate how STREAM eliminates such redundancy through *synopsis sharing*.

4.3 Concepts

In this section, we give a more thorough presentation of the most important concepts in STREAM.

4.3.1 Internal Representation of Streams and Relations

Recall the formal definitions of streams and relations in Section 4.1.1. A stream is a bag of tuple-timestamp pairs, which may be expressed as a sequence of timestamped tuple *insertions*. A relation, which is a time-varying bag of tuples, can also be represented as a sequence of timestamped tuples, except now we have both *insertion tuples* and *deletion tuples* to capture the changing state of the relation (Arasu et al. 2003b). In the implementation, these two types are unified as a sequence of timestamped tuples, where each tuple additionally is flagged as either an insertion (+) or deletion (-). The tuple-timestamp-flag triples are referred to as *elements* (Arasu et al. 2004a). Streams only include + elements, while relations may include both + and – elements.

4.3.2 Query Plans

A query plan in STREAM runs continuously and is, as we mentioned in Section 4.2.3, composed of three different types of components: operators, queues, and synopsis. When a query plan is executed, a *scheduler* selects operators in the plan to execute in turn. In this section, we provide a more thorough description of the query processing architecture of STREAM.

Operators

Query operators correspond to the three types of operators in the abstract semantics (Section 4.1.2). Each query plan operator reads from one or more *input queues*, processes the input based on its semantics, and writes any output to an *output queue* (Arasu et al. 2004a). Since queues encode both streams and relations, query plan operators can implement all three operator types in the abstract semantics and CQL namely stream-to-relation, relation-to-relation, and relation-to-stream (Arasu et al. 2003b).

Queues

A queue in a query plan connects its input operator O_I to its output operator O_O. At any given time, a queue contains a (possibly empty) sequence of elements representing a portion of a stream or relation. The elements that O_I produces are inserted into the queue and are buffered there until they are processed by O_O (Arasu et al. 2004a; Arasu et al. 2003b). Many of the operators in STREAM require that elements on their input queues is read in non-decreasing timestamp order. Consider, for example, a window operator O_W on a stream *S*. If O_W receives an element $\langle s, \tau, + \rangle$ and its input queue is guaranteed to be in non-decreasing timestamp order, than O_W knows it has received all elements with timestamp $\tau' < \tau$, and it can construct the state of the window at time $\tau - 1$. If, on the other hand, O_W does not have this guarantee, it can never be sure it has sufficient information to construct any window correctly. Thus, STREAM requires all queues to enforce non-decreasing timestamps (Arasu et al. 2004a).

Synopses

Logically, a synopsis belongs to a specific plan operator, storing state that may be required for future evaluation of that operator. For example, to perform a windowed join of two streams, the join operator must have access to all tuples in the current window on each input stream. Thus, the join operator maintains one synopsis (e.g. a hash table) for each of its input (Arasu et al. 2004a; Arasu et al. 2003b). On the other hand, operators such as selection and duplicate-preserving projection do not require any synopses. The most common use of a synopsis in STREAM is to materialise the current state of a (derived) relation, such as the contents of a sliding window or the relation produced by a sub-query (Arasu et al. 2004a).

4.3.3 Performance Issues

Straightforward generation and execution of query plans can be very inefficient. Several techniques to improve performance are available such as the Eddy in TelegraphCQ (Avnur et al. 2000). In this section, we present techniques for performance improvement implemented in STREAM.

Resource Sharing

When continuous queries contain common sub-expressions, it is possible to share resources and computation within their query plans. Two such resource-sharing techniques are *queue sharing* and *synopsis sharing*.

Queue Sharing

The implementation of a shared queue maintains a pointer to the first unread tuple for each operator that reads from the queue, and it discards tuples once they have been read by all parent operators (Motwani et al. 2003). The number of tuples in a shared queue at any time depends on the rate at which tuples are added to the queue, and the rate at which the slowest parent operator consumes the tuples. If two queries have common queues, and the two distinct output operators to those queues have very different consumption rates, it may be preferable not to utilise a shared queue.

Synopsis Sharing

In Section 4.2.3, we observed that multiple synopses within a single query plan might materialise nearly identical relations. By introducing a single *store* to hold the actual tuples, and replacing the two synopses with lightweight *stubs*, STREAM is able to eliminate such redundancy. These stubs implement the same interface as non-shared synopsis. Thus, operators can be ignorant to the details of sharing, and as a result, synopsis sharing can be enabled or disabled on the fly (Arasu et al. 2004a).

When synopses are shared, logic in the store tracks the progress of each stub, and presents the appropriate view (subset of tuples) to each of the stubs. The store contains the union of its corresponding stubs: A tuple is inserted into the store as soon as it is inserted by any of the stubs, and it is removed only when it has been removed from all the stubs.

To further decrease state redundancy in STREAM, multiple query plans involving similar intermediate relations can share synopses as well. For example, suppose the following query is registered in addition to the query in Section 4.2.3:

```
SELECT A, MAX(B) FROM S1 [ROWS 200] GROUP BY A
```

Since sliding windows are contiguous in STREAM, the window on S1 in this query is a subset of the window on S1 in the other query. Thus, the same data store can be used to materialise both windows. The combination of the two query plans with both types of sharing is illustrated in Figure 12 below adapted from Arasu et al. (2004a).

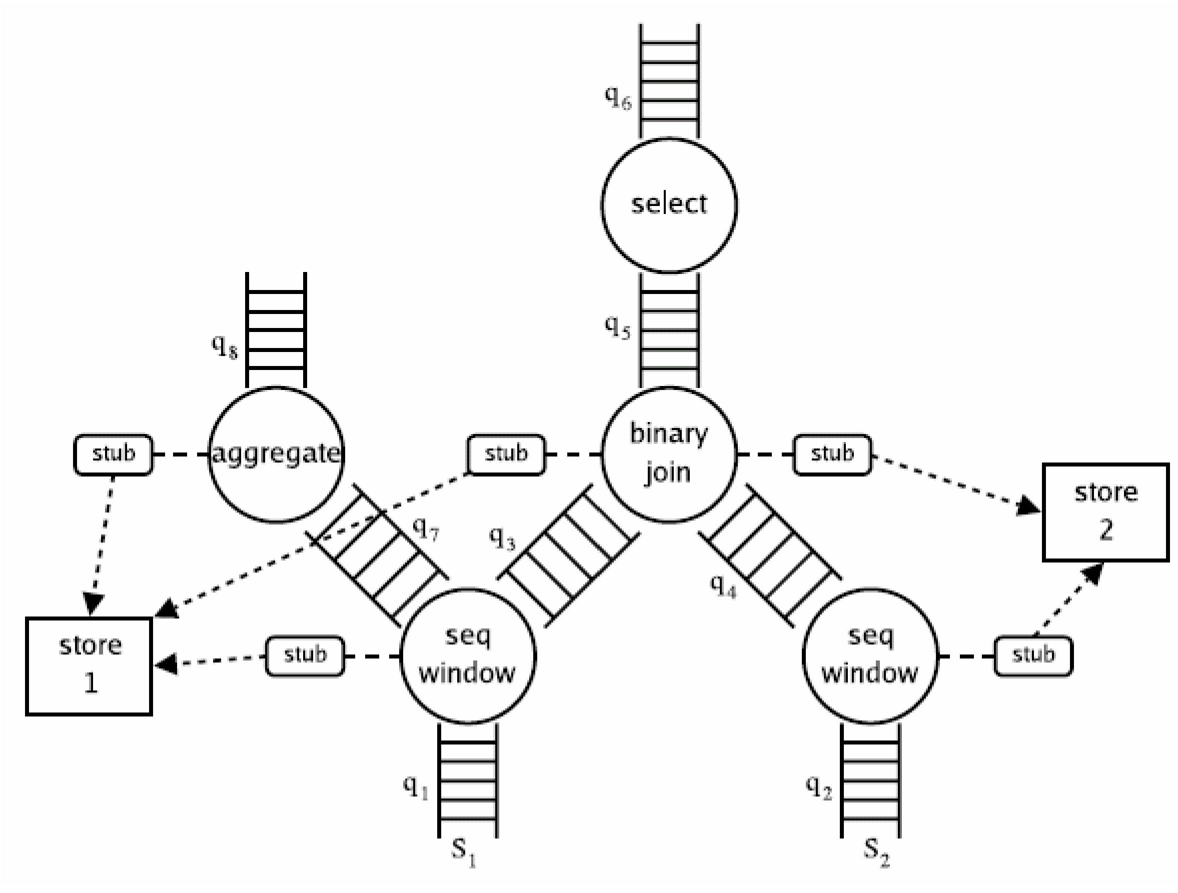


Figure 12. A query plan illustrating synopsis sharing

Exploiting Constraints

Streams may contain data or arrival patterns, which are possible to explore in order to reduce run-time synopsis sizes. Such *constraints* can either be specified at stream registration time, or by gathering statistics over time (Arasu et al. 2004a; Arasu et al. 2003b).

In STREAM, they have identified several types of useful constraints over data streams. The constraints they have considered in their work are *many-to-one join* and *referential integrity* constraints between two streams, and *clustered-arrival* and *ordered-arrival* on individual streams (Arasu et al. 2003b). To make effective optimisation possible, even when the constraints are not strictly met, they have defined an *adherence parameter k* that captures how closely a given stream or pair of streams adheres to a constraint of that type. They refer to these as *k-constraints* (Arasu et al. 2004a). As the value of *k* for each constraint becomes smaller, more and more state can be discarded. A thorough description of *k*-constraints is presented by Babu et al. (2004a).

Operator Scheduling

An operator consumes elements from its input queues and produces elements on its output queue. *Selectivity* is the ratio, over one time unit, between the memory size an operator processes from its input queues, and the memory size it produces on its output queue. The selectivity *s* is at most one for the select and project operators, but it may be greater than one for a join. Depending on the selectivity of an operator, the scheduling algorithm having the best performance may vary in different input stream scenarios. Examples of such scenarios are presented by Arasu et al. (2004a) and Babcock et al. (2003). Due to the uncertainty in which algorithm performs best, they develop a new scheduling algorithm, named *chain scheduling*, in STREAM. This algorithm forms blocks ("chains") as follows: Start by marking the first operator in the query plan as the "current" operator. Next, find the block of consecutive operators starting at the "current" operator that maximises the reduction in total queue size per time unit (smallest selectivity). Mark the first operator following this block as the "current" operator and repeat the previous step until all operators are assigned to chains. Chains are scheduled according to the greedy algorithm, but within a chain, execution proceeds in FIFO order (Arasu et al. 2004a). Further descriptions of chain scheduling are presented by Babcock et al. (2003).

4.3.4 Adaptivity

Over time, data and arrival characteristics of a stream may vary significantly. In addition, query loads and system conditions may vary. Therefore, an *adaptive* approach to query processing is important to prevent performance to drop drastically as the environment change. The STREAM system includes a monitoring and adaptive query-processing infrastructure called StreaMon. StreaMon has three components as shown in Figure 13 below adapted from Arasu et al. (2004a). Firstly, the *Executor*, which runs query plans to produce results, secondly, the *Profiler*, which collects and maintains statistics about streams and query plan characteristics, and thirdly, the *Reoptimiser*, which ensures that the plans and memory structures are the most efficient for current characteristics. A thorough description of StreaMon is provided by Babu et al. (2004b).

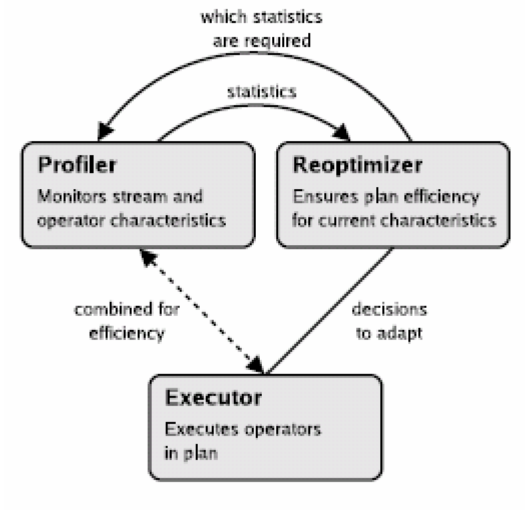

which statistics are required
statistics
Profiler
Monitors stream and operator characteristics
Reoptimizer
Ensures plan efficiency for current characteristics
combined for efficiency
decisions to adapt
Executor
Executes operators in plan

Figure 13. StreaMon

4.3.5 Approximation

In many applications, data streams arrive with bursts that may cause unpredictable peaks during which the load may exceed available system resources. For many applications it is acceptable to degrade accuracy by providing approximate answers during such load spikes. A DSMS may be resource limited in two primary ways. It may be *memory-limited* or *CPU-limited* (Arasu et al. 2004a).

Memory-Limited Approximation

When the total state required for all registered queries exceeds available memory, the DSMS becomes memory-limited. Recall that STREAM's chain scheduling algorithm is greedy (when scheduling chains) based on selecting operators that maximises the reduction in total queue size. In addition, the constraint-aware execution strategy, which was described in Section 4.3.3, minimises the synopsis sizes. A thorough description of this topic is presented by Babu et al. (2004a). Even by the use of these memory-reducing strategies, memory may become limited. In such a scenario, memory usage can be reduced at the cost of accuracy by reducing the size of synopses at one or more operators. Examples of methods for reducing synopsis size are maintaining a sample of the intended synopsis content, using histograms or wavelets when the synopsis is used for aggregation or even for a join, and using Bloom filters for duplicate elimination (Arasu et al. 2004a). However, other memory reducing methods than chain scheduling and exploiting k-constraints are not implemented in the current STREAM prototype.

CPU-Limited Approximation

When the data arrival rate is so high that there is insufficient CPU time to process each stream element the DSMS becomes CPU-limited. CPU usage can be reduced by *load shedding* i.e. dropping elements from query plans and saving the CPU time that would be required to process them to completion (Arasu et al. 2004a). In STREAM, they implement load shedding by introducing *sampling* operators that may drop stream elements as they are input to the query plan. Load shedding is introduced at the cost of accuracy, however, it is not implemented in the current STREAM prototype.

4.4 How to Use STREAM

In this section, we illustrate how to use the STREAM by providing a brief description of the most important concepts. A more detailed explanation of how to use the system is given in the STREAM user guide and design document (Anonymous 2004). We start by describing the generic command-line client, gen_client. Secondly, we describe the different arguments that are necessary when running gen_client. Finally, the sources of the streams and relations involved in queries will be considered.

4.4.1 Gen_client

When running STREAM, we use the generic command-line client, gen_client, which use the STREAM library. Briefly, the steps involved in using STREAM as an embedded library are; registering a new Server object, configuring the server by providing a configuration file, registering the input streams and relations, registering the queries, generating a query plan, and starting the server. The usage syntax of the gen_client program is:

```
gen_client -l [log-file] -c [config-file] [script-file]
```

[log-file] is the output file where the execution log of the program is written, and this file is created if it does not already exist. [config-file] is an input file that specifies values of various server configuration parameters, whereas, [script-file] is an input file that contains the queries to be executed, and the streams and relations involved in the queries.

4.4.2 The Configuration File

The configuration file parameter to gen_client contains values that configure STREAM. Throughout our analysis, we use one configuration file. We have changed the values for memory size and run time compared to the example configuration file that is included in the STREAM prototype release.

```
MEMORY_SIZE = 805306368

QUEUE_SIZE = 30

SHARED_QUEUE_SIZE = 300

INDEX_THRESHOLD = 0.85

RUN_TIME = 10000
```

None of the configuration parameters are strictly necessary. If the value for some parameter is not specified, the system assumes default values. `MEMORY_SIZE` contains the size of the memory in bytes that is used during the execution of the system. 805 306 368 bytes equals 768 MB. `QUEUE_SIZE` contains the size of the queues, which have fixed sizes. A smaller value of `QUEUE_SIZE` means that the operators execute in a more tightly coupled manner. This should be an integer value larger than one. The queue size is specified in number of pages. A page is set to 4096 bytes. `SHARED_QUEUE_SIZE` contains shared queue sizes in pages. A shared queue is a queue that has one write operator and many read operators. It is useful to set this value higher than `QUEUE_SIZE`. `INDEX_THRESHOLD` should be a fraction between zero and one. It is similar to the threshold value used in a disk-based linear hash table. When the ratio between the number of tuples and the number of buckets exceeds the threshold, buckets are added and the table is rehashed. A smaller value leads to cheaper index updates but lookups could be costlier and vice-versa. `RUN_TIME` is the number of times STREAM receives empty tuples from the table source before exiting.

4.4.3 The Scipt File

In the current STREAM prototype, four data types are implemented. These are `integer`, `float`, `char(n)`, and `byte`, where the *n* in char(n) is the length of the string in bytes, including the string terminator. In the following example, we only show how to register a stream and how to query it.

```
Table : register stream S (A integer, B float, C
char(10));

Source : hernes/examples/Example.dat

Query : select B, C from S where A < 20;

Dest : hernes/examples/ExampleOutput
```

The first line in this example defines the table's schema. If a relation is registered, the word "stream" is replaced with "relation". The next line provides the location of the source file. The actual query is given in the line that starts with "Query", and the destination for the file containing the query answer is given in the line starting with "Dest". One should realise that each statement must be given in one line without a line break. The STREAM user guide and design document (Anonymous 2004) gives a detailed descriptions on how to for example define windows and views. In addition, the document explains several semantic similarities.

4.4.4 Table Source

As we described in the previous section, the word "stream" can be replaced with the word "relation" when defining the table's schema. However, the source file representing a stream is different from a source file representing a relation. In this section, we give a brief description of their characteristics.

Stream Source

The file representing the stream contains the stream's tuples. However, the first line must be a description corresponding to the tables schema. For the example in the previous section, the first lines may look like:

```
i, i, f, c10
0, 1, 1.11, text1
1, 2, 2.22, text2
```

The “i” means integer, “f” means float, and “c10” means a string of ten bytes. The first “i” represents the timestamp. The timestamp is not an attribute and may not be queried.

Relation Source

If replacing the word “stream” with “relation” in the query example, the file representing the relation may look like:

```
i, b, i, f, c10
0, +, 1, 1.11, text1
0, +, 2, 2.22, text2
```

In a DBMS, deleted tuples are not registered within a relation; they are simply removed. Thus, a relation stored on file that is registered within STREAM should normally only contain insertions although it is possible to let the file contain deletions as well. However, the deletions have most value in the intermediate relations or when writing a relation to file to track changes in the relation. Notice that the relation’s tuples contains timestamps. The timestamps too have most value in the intermediate relations and when tracking changes in the relation. However, the timestamps introduce a problem when joining a relation stored on file with a stream. Recall from Section 4.3.2 that when an operator receives an element with timestamp *t*, it knows it has received all elements with timestamp *t’* < *t*. If each tuple in the relation should be joined with each tuple in a window over the stream, they should all have zero as timestamp, at least if they should be joined with the tuples in the stream that also have zero as timestamp. We assume that the stream is long lasting with a wide range of timestamps in increasing order. The problem is that the join operator cannot join any of the tuples from such a relation before it has seen a tuple, from the relation, with a timestamp larger than zero. If such a tuple is inserted, this tuple will not be joined before a timestamp larger than this tuple’s timestamp is seen. Thus, when defining relations we use a dummy-tuple as the last tuple. This tuple has a very large number as timestamp, thus, the tuple will never be a part of the join. However, it unblocks the rest of the relation. Such a tuple takes the form of a heartbeat in the relation. Heartbeats are not in the scope of this thesis, however, a thorough description of heartbeats are provided by Srivastava et al. (Srivastava et al. 2004).

5. Query Design

In this section, we consider different network monitoring tasks and design queries solving them. We will use some of these queries in our performance evaluation in Section 7. The design of the queries is affected by possibilities and limitations of the current STREAM implementation. As mentioned in Section 4, we use the latest STREAM implementation, STREAM 0.6.0, in our work with this thesis.

Prior to designing the different queries, we perform some analysis regarding which data types to use when defining attributes of network data. In addition, we define an input stream that we use throughout the rest of the thesis. After defining the input stream, we design queries solving the different network monitoring tasks.

5.1 The Input Stream S

In all tasks, outlined later in this section, we use a generic input stream S, which consists of TCP/IP packet header values. In S, one tuple contains values from all TCP and IP header fields. Figure 14 below illustrates the TCP header format.

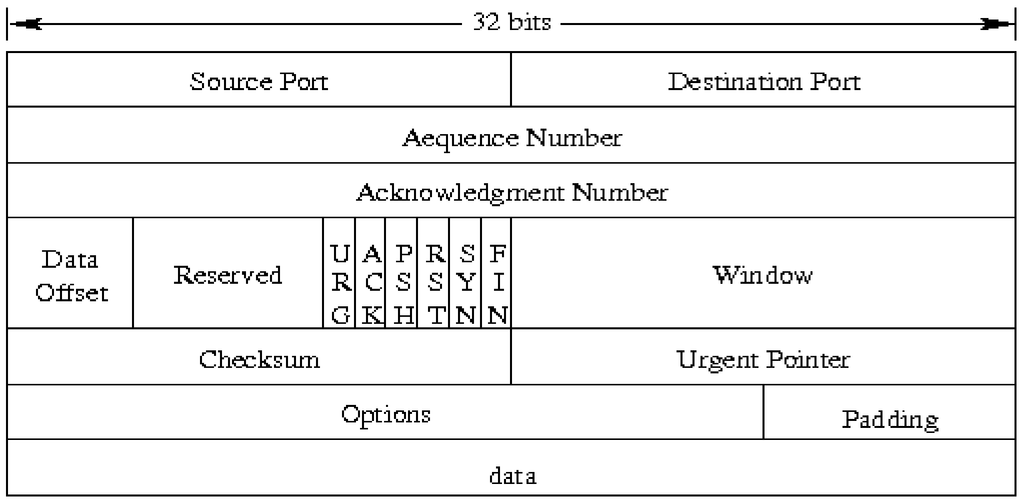


Figure 14. TCP header format

The IP header format is shown in Figure 15 below. S only contains header information from packets where IP headers encapsulate TCP packets. Header fields from the MAC layer are not included. The reason for this is that we do not consider any network monitoring tasks that analyse header fields from this layer.

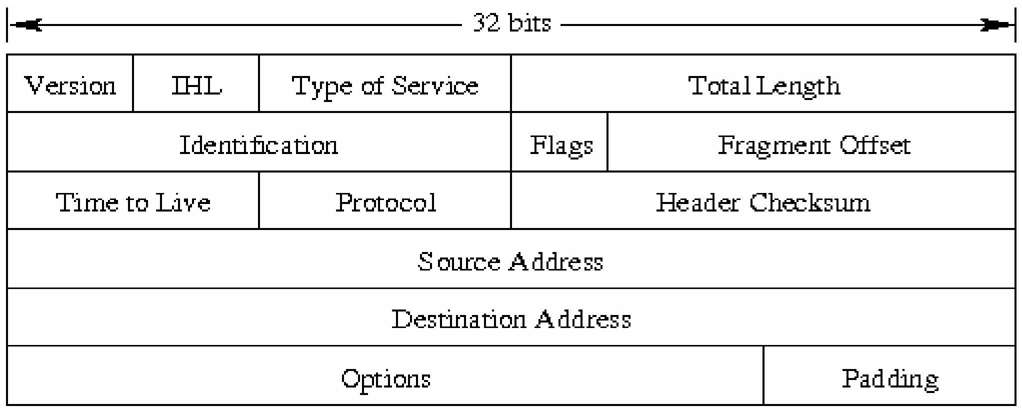


Figure 15. IP header format

In STREAM, no data types may directly represent IP addresses or the TCP control flags URG, ACK, PSH, RST, SYN, and FIN. Consequently, we perform some analyses in order to decide which data type to use when defining these attributes. We use several different queries in these analyses. Each query is executed offline, and repeated ten times for each representation. We execute the queries on computers B, which will be described in Section 6.3.2. In addition to these analyses, we discuss how we represent the TCP and IP option fields.

5.1.1 IP Addresses

In IP headers there are two fields containing IP addresses. There is one field for source IP address, and one field for destination IP address. In STREAM, there is no data type made explicitly for representing IP addresses such as `cidr` in PostgreSQL. Hence, either `integer` or `char(n)` must be used to present these addresses. For example, the IP address 10.10.10.10 may be represented in three different ways:

1. As four integers: 10, 10, 10, 10
2. As four char(4): 10, 10, 10, 10
3. As one char(16): 10.10.10.10

There is also a fourth representation, namely the whole address as an `unsigned integer`. This is possible since both integers and IP addresses use 32 bits. The example address is in this case written as 168430090. Normally, the 32-bits address is split into four parts of one byte. Consequently, we prefer one of the three representations in the list.

In the Internet, data streams arrive with bursts, and the load may exceed available system resources. The data arrival rate may be so high that there is insufficient CPU time to process each stream element. In this case, the system may drop elements before they are processed and, thus, provide approximate query answers. Though it in many streaming applications is acceptable to degrade accuracy by providing approximate answers during load spikes (Arasu et al. 2004a), it is essential to keep processing time as low as possible. During load spikes, the system may also become memory limited. The total state required for all registered queries may exceed available memory. Consequently, it is important to keep memory usage to a minimum, favouring those representations requiring the least number of bytes. A couple of bytes saved on each tuple, may result in several MB saved in large windows. Hence, *processing time* and *memory usage* are the main metrics when deciding which of the three IP address representations to choose. However, each representation requires 16 bytes to represent an IP address. This makes each representation equally suitable when compared to memory usage. The representation resulting in the lowest processing time may vary from query to query. Therefore, we construct different types of queries in this analysis. In addition to processing time and memory usage, we compare representations in how *applicable* they are i.e. each representation is measured in the availability of operators that we consider necessary when solving network traffic monitoring tasks.

Processing Time

All queries consider an input stream of the same format as S. However, in this analysis we do not stream real internet traffic into the system. Instead, we create some files containing dummy data. The two first representations in the list above require four attributes for each IP address. To represent source IP addresses, we use `sourceA`, `sourceB`, `sourceC`, and `sourceD`, while we use `destA`, `destB`, `destC`, and `destD` to represent destination IP addresses. For the last representation, it is sufficient with `sourceIP` and `destIP`. We only show queries made for the last representation. In the queries, `sourceIP` and `destIP` may be replaced with the first attributes. The first query, Query1, projects source IP addresses.

```
SELECT sourceIP
FROM S
```

The next query, Query2, groups by source IP address and destination IP addresses, and for each such combination it counts the number of packets i.e. the number of packets sent from a source to a destination. Query2a uses a one-second window, while Query2b uses a window size of ten seconds. We only present Query2a here.

```
SELECT sourceIP, destIP, count(*)
```

```
FROM S [RANGE 1 SECOND]
GROUP BY sourceIP, destIP
```

In the last query, Query3, we introduce a second stream, T, which has the same attributes as S. This query finds, for all packets in S and T, the total number of bytes sent to each destination IP address. First, we create one view querying S, and one view querying T. Each view selects, for each packet, the destination IP address, and the total length. A third view selects the union of the two previous views. In the final query, we query the third view grouping by destination IP, and summarising the number of bytes. In Query3a, this is done over a one-second window, while Query3b uses a window size of ten seconds. We only present Query3a here.

```
VS (destIP, totalLength):
SELECT destIP, totalLength
FROM S

VT (destIP, totalLength):
SELECT destIP, totalLength
FROM T

VU (destIP, totalLength):
VS UNION VT

RSTREAM(
SELECT destIP, sum(totalLength)
FROM VU [RANGE 1 SECOND]
GROUP BY destIP)
```

When creating S and T, we randomly choose IP addresses in a range from 10.10.10.0 to 10.10.10.255. This address range corresponds to an eight-bit subnet. Both streams simulate a rate of 50 packets per second, over a total of 900 seconds, or 15 minutes. This results in 45.000 packets in each stream. We use the `time` command in Linux to monitor for how long STREAM has been processing. Average processing time for these ten executions is presented in Table 1 below. The processing time is given in seconds. The table shows that the third representation uses least time to process its input streams.

	Representation 1	Representation 2	Representation 3
Query 1	1.0009	0.9773	0.9233
Query 2a	1.3177	1.3741	1.2304
Query 2b	1.3529	1.3889	1.2585
Query 3a	1.9467	2.0364	1.8834
Query 3b	2.9247	3.0453	2.7916

Table 1. Average processing time in seconds when processing streams with 15 minutes duration.

We perform another round of executions with new files simulating the input streams. In these files, the streams simulate a rate of 100 packets per second, over a total of 1800 seconds, or 30 minutes. This results in 180.000 packets in each stream. Average processing time for these ten executions is presented in Table 2 below. This table shows an equal tendency as in Table 1, with the third representation using least time to process its input streams.

	Representation 1	Representation 2	Representation 3
Query 1	5.7312	5.7451	5.2747
Query 2a	8.0310	8.2128	7.0967
Query 2b	8.2075	8.4169	7.3013
Query 3a	10.8878	11.2545	10.2540
Query 3b	14.7400	15.2741	13.8217

Table 2. Average processing time in seconds when processing streams with 30 minutes duration.

Applicability

One obvious restriction of the two last representations is that attributes of data type `char(n)` cannot be included in arithmetic expressions. However, we do not consider such operations necessary regarding IP addresses. In addition, there are restrictions to using `char(n)` when applying logical operators. For example, consider what happens if we have a ten bits subnet with addresses between 10.10.0.0 and 10.10.3.255, and the third representation of IP addresses. The `WHERE` clause in a query checking if a destination IP address belongs to this subnet may look like:

```
WHERE destIP >= "10.10.0.0" AND destIP <= "10.10.3.255".
```

This statement would not only select correct IP addresses, because of how strings are compared e.g. 10 is smaller than 2. Mistakenly, an address like e.g. 10.10.10.10 would be considered a member of the given subnet. The only logical operators that give correct results when comparing IP addresses, defined with `char(n)`, are `!=` and =. The only task that could make use of subnet matching is Task 6, which will be discussed in Section 5.2.6.

Conclusion

When measuring processing time, the third representation, where IP addresses are represented with `char(16)`, is most suitable. However, when comparing how applicable the representations are, the most suitable solution is the first representation where IP addresses are represented with four `integers`. We choose the third representation of IP addresses due to the following two reasons. Firstly, the difference in how applicable the representations are does not become visible in any of the tasks considered in this document and, secondly, we consider processing time to be a more important metric.

5.1.2 Control Flags

A TCP header contains six control flags: URG, ACK, PSH, RST, SYN, and FIN. Each flag uses one bit to indicate whether it is set. The smallest data type in STREAM is `B`, which occupies one byte. However, STREAM is not implemented to operate on this data type. Consequently, the control flags can be defined using `integer` or `char(2)`. When analysing the performance of these two representations we use the same metrics as in the analysis of IP addresses. The only values control bits may have are zero or one, because they are represented by one bit. Consequently, the only necessary operation on these attributes is to check whether they are set e.g. `SYN = "1"`, or `SYN = "0"`, assuming `char(2)` representation. No arithmetic operations are necessary, hence, both representations are equally suitable with respect to applicability.

Memory Usage

The `char(2)` representation occupies two bytes, while the `integer` representation occupies four bytes. The two-bytes difference between these representations results in 12 bytes difference for all control bits in a tuple. To illustrate the magnitude of these 12 bytes, consider an Ethernet with constant traffic at a bit rate of 100 Mb/s, and all packets carrying the Ethernet's Maximum Transfer Unit (MTU) with a packet size of 1500 bytes (12 000 bits). The arrival rate on the Network Interface Card (NIC) is then 8 738.1333 packets per second. With 12 bytes (96 bits) per packet this equals 104 857.6 bytes per second. With a window size of one minute, this results in 6 MB.

Processing Time

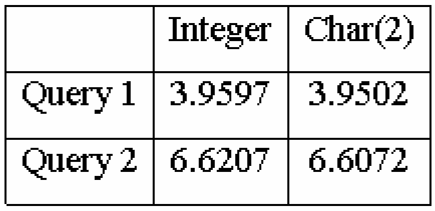
To compare processing time between the two different representations, we consider two queries, Query 1 and Query 2. The first query selects all control flags from the input stream.

```
SELECT URG, ACK, PSH, RST, SYN, FIN
FROM S
```

In the second query, TCP connections are recognised, and a detailed discussion of this query will be described in Section 5.2.4. This query consists of two views, both querying S, and a final query joining these views with S. The final query recognises connections over a three-minute window.

```
SYN (sourceIP, destIP, sourcePort, destPort, seqNum,
ackNum):

SELECT sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum

FROM S

WHERE SYN = "1" AND ACK = "0"


SYNACK (sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum):

SELECT sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum

FROM S

WHERE SYN = "1" AND ACK = "1"


SELECT DISTINCT SYN.sourceIP, SYN.destIP,
SYN.sourcePort, SYN.destPort

FROM SYN [RANGE 3 MINUTES], SYNACK [RANGE 3
MINUTES], S [RANGE 3 MINUTES]

WHERE SYN.sourceIP = SYNACK.destIP AND SYN.sourceIP
= S.sourceIP AND SYN.destIP = SYNACK.sourceIP AND
SYN.destIP = S.destIP AND SYN.sourcePort =
SYNACK.destPort AND SYN.sourcePort = S.sourcePort
AND SYN.destPort = SYNACK.sourcePort AND
SYN.destPort = S.destPort AND SYN.seqNum + 1 =
S.seqNum AND SYNACK.seqNum + 1 = S.ackNum
```

We use an input stream similar to S in the execution of both queries. We create S to simulate a stream with an arrival rate of 100 packets per second, with a 30 minutes duration. Three of the 100 packets are used to establish a connection, which means that one connection is established every second. Each query is executed ten times for each representation, and we calculate the average processing time for the ten executions. Table 3 below presents the average processing time. It shows that STREAM processes the queries slightly faster when using `char(2)` rather than `integer`.



	Integer	Char(2)
Query 1	3.9597	3.9502
Query 2	6.6207	6.6072

Table 3. Average processing time in seconds when processing streams with 30 minutes duration.

Conclusion

The representation using `char(2)` has shortest processing time, and occupies less memory than the representation using `integer`. Consequently, we choose `char(2)` to represent the control flags.

5.1.3 Option Fields

The IP and TCP headers include one field each that describe the length of the header in 32-bits words. These fields are known as Header Length and Data Offset, respectively, and they consist of four bits. That results in a maximum header length of 15 i.e. 15 words of 32 bits. Since five is the minimum length of both headers, there are ten 32 bits words left for the option field. In order to represent the option field as one attribute, we may only use the data type `char(n)`. Consequently, we give the option fields in hexadecimal numbers. Then 80 hexadecimal numbers are required to represent the possible 320 bits in an option field. In STREAM, we require one character for each hexadecimal number, in addition to the string terminator. Thus, option fields are defined as `char(81)`.

5.1.4 Definition of Stream S

A schema over S is defined with the following data definition language (DDL) statement:

```
REGISTER STREAM S(
version integer,
ip_headerLength integer,
tos integer,
totalLength integer,
id integer,
flags integer,
fragOffset integer,
ttl integer,
protocol integer,
headerChecksum integer,
sourceIP char(16),
destIP char(16),
ip_options char(81),
sourcePort integer,
destPort integer,
seqNum integer,
ackNum integer,
tcp_headerLength integer,
reserved integer,
URG char(2),
ACK char(2),
PSH char(2),
RST char(2),
SYN char(2),
    FIN char(2),
    window integer,
    checkSum integer,
    urgent integer,
    tcp_options char(81))
```

5.2 Solving Network Monitoting Tasks

In this section, we design queries solving the different network traffic monitoring tasks. The four tasks discussed in the sections from Section 5.2.3 to Section 5.2.6 are adapted from Plagemann et al. (2004), where they were solved in TelegraphCQ. For these tasks, we consider whether it is possible to use similar solutions within STREAM. For the remaining tasks, we discuss whether they are possible to solve online with continuous queries in STREAM. If they cannot be solved with continuous queries, we will attempt to explain why. In addition, prior to presenting the different solutions here, we test the queries offline by processing data stored in files. Thus, all queries presented in this section are accepted by STREAM and produces the query answer that is expected based on the data to be queried. All tasks assume a generic input stream S, which was defined in the previous section.

In Section 4.4.3, we described the script file used to define sources, queries, and destinations. A script-based language is used when implementing the script file. However, to make the queries more readable, we maintain syntax as close as possible to SQL in this section. The syntax is adopted from the STREAM Query Repository (Anonymous 2002) and the STREAM User Guide and Design Document (Anonymous 2004). The largest syntactic difference between the adopted syntax and the syntax in the script file is found in the creation of views.

As we explained in Section 4.1.3, CQL includes stream-to-relation operators, which are used to unblock blocking queries. As the name suggests, a stream is converted into a relation. Then relation-to-relation operators are executed on the (derived) relation, with a relation as the result. If desirable, this relation may be transformed back into a stream, using one of the relation-to-stream operators: `Istream`, `Dstream`, or `Rstream`.

When referring to the different tasks, we use the word “Task” followed by a number. This number agrees with the paragraph number of the section describing the task e.g. the task described in Section 5.2.1 is referred to as Task 1 and the task described in Section 5.2.2 is referred to as Task 2.

5.2.1 What is the average network load measured in bytes per second?

To calculate bytes per second, we summarise the total length from all headers within a one-second window. We then calculate the average of this sum. Thus, one may expect that this solution require a nested aggregation. Nested aggregation is one of the requirements that were discussed in Section 2.3. However, STREAM does not support nested aggregation as defined there. Besides, an aggregating operator is blocking and must therefore be calculated within a window. Unless used in relation to the GROUP BY clause, the `SUM` operator in a nested aggregation e.g. `AVG(SUM(total length))`, produces one tuple within a window. The average produced by the `AVG` operator is then equal to the sum, because the average is calculated over one tuple. Thus, when using the `AVG` operator to calculate the average, we first calculate the sum in a view using a window size of one second. Then, we query this view with a larger window size to calculate average network load. For statistics, average can be calculated over a day, or a week. In such cases, and if storage is not a limitation, this task is probably better-solved offline with the well-established DBMS technology. If we want to respond as fast as possible to situations in which average grows over a certain threshold, a smaller window should be used. In such scenarios, a suitable window size may be five, or ten seconds. In our solution, we use a window size of ten seconds. Thus, first we create a view that selects bytes per second. This view is queried to calculate average network load over the ten last seconds. To stream the state of the intermediate relation as the window slides, we encapsulate the view and the final query by `Rstream.`

```
Load (bytesPerSec):
RSTREAM(
SELECT SUM(totalLength)
FROM S [RANGE 1 SECOND])

RSTREAM(
SELECT AVG(bytesPerSec)
FROM Load [RANGE 10 SECONDS])
```

5.2.2 What is the network load measured in packets per minute?

This task requires a similar solution as the previous one. The query solving the task first creates a view where packets are counted over a one-minute window. Next, this view is queried in order to calculate the average load. Once more, a suitable window size when calculating average may be ten seconds.

```
Load (packetsPerMinute):
RSTREAM(
SELECT COUNT(*)
FROM S [RANGE 1 MINUTE])

RSTREAM(
SELECT AVG(packetsPerMinute)
FROM Load [RANGE 10 SECONDS])
```

5.2.3 How many packets have been sent during the last five minutes to certain ports?

In order to solve this task, we first create a relation containing ports of interest. What ports being of interest may vary, depending on the monitoring application. The query is an example of a join operation between a stream and a relation. A solution similar to the one suggested by Plagemann et al. (2004) would in STREAM look like:

```
REGISTER TABLE Services (port integer);


SELECT R.port, COUNT(*)
FROM S [RANGE 5 MINUTES], Services AS R
WHERE S.destPort = R.port
GROUP BY R.port
```

However, this solution does not work within STREAM. Tests show that it is not possible to aggregate over joins where the total amount of attributes from the sources is more than 20. The total number of attributes in the join between S and Services is 30. S has 29 attributes, and Services has one. Hence, we rewrite the query to solve this task. In order to reduce the number of attributes, we first create a view, DestPorts, where only destination port numbers are projected from the stream. This view is not blocking, and the packets simply stream from the input queue to the output queue.

```
DestPorts (destPort):
SELECT destPort
FROM S
```

We use the same definition of Services as identified above. Finally, DestPorts is joined with Services in order to find the number of packets that have been sent during the last five minutes to certain ports.

```
RSTREAM(
SELECT R.port, COUNT(*)
FROM DestPorts [RANGE 5 MINUTES] AS P, Services AS R
WHERE P.destPort = R.port
GROUP BY R.port)
```

5.2.4 How many bytes have been exchanged on each connection during the last minute?

In the solution proposed by Plagemann et al. (2004), a simple heuristic was employed in order to define a connection. During a one-minute window, all packets with the same sender and receiver IP addresses and port numbers belong to the same connection. Consequently, the query contained the `GROUP BY` clause followed by the quadruple `sourceIP`, `sourcePort`, `destIP` and `destPort` as grouping attributes over a one minute window. However, a TCP connection has two sides, a client, and a server. The values in the quadruple at one side is opposite on the other side e.g. the client's IP address is `sourceIP` when the client sends packets, and `destIP` when the server sends packets. Hence, the solution proposed by Plagemann et al. (2004) defines a connection as data moving, in one direction from one port to another. In addition, their solution does not consider that some of the TCP connection initiatives may not lead to established connections e.g. SYN flood attacks. However, recognising established connections requires a series of sub-queries, and this feature was not supported in the TelegraphCQ prototype.

We propose a solution, which takes into account that connections are established through the accomplishment of a three-way-handshake (Postel 1981b; Tanenbaum 2003). In addition, the solution defines a connection to contain data flowing in both directions between a pair of port numbers, represented by the quadruple `sourceIP`, `sourcePort`, `destIP`, and `destPort`. Since a connection has different representations, depending on which direction the data flows, we choose to represent a connection by values collected at the client's side.

As mentioned above, a connection is established through a three-way handshake as illustrated in Figure 16 below. The client opens the connection through an *active open*, while the server opens it through a *passive open*. The client sends a SYN message i.e. a message with the SYN flag set. This message has the sequence number `x`. Upon receiving this message the server responds with a SYN message containing sequence number `y`, and an ACK message containing acknowledgment number `x +`

`1`. However, as an optimising effort these two messages are combined in one, a SYN/ACK message i.e. a message with both the SYN flag and the ACK flag set. Finally, upon receiving the SYN/ACK message, the server sends an ACK message i.e. a message with the ACK flag set. This message has the sequence number `x + 1` and the acknowledgement number `y + 1`.

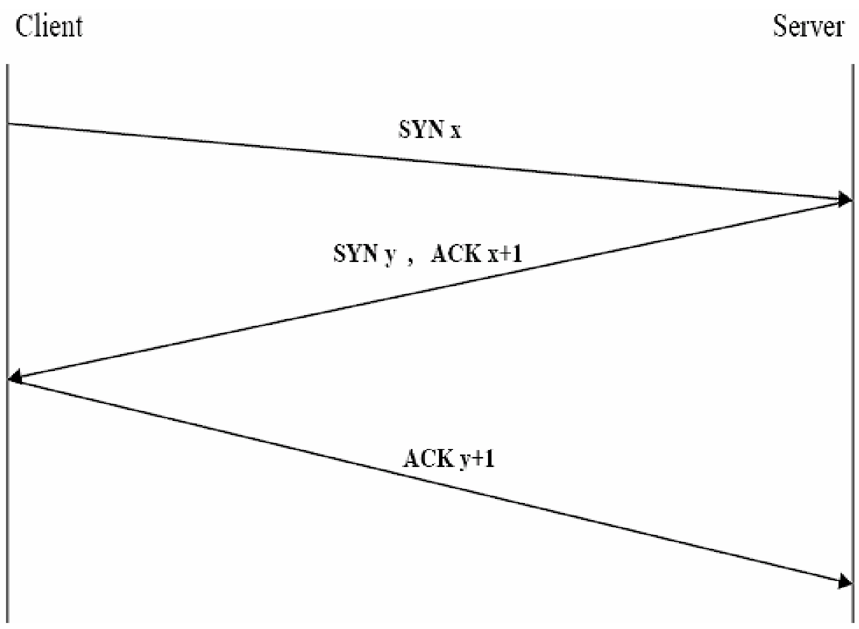


Figure 16. TCP connection establishment

To identify connections, we first create two different views where SYN, and SYN/ACK, messages in S are identified. Based on equality of IP addresses, port numbers, sequence numbers, and acknowledgement numbers, these views are joined with S in order to identify ACK messages corresponding to the third step in the handshake. This information is collected in a view, Conn, where each tuple represent an established connection. The data flow in our solution is portrayed in Figure 17 below. Rectangular boxes represent views. The lowest rectangular box, with rounded corners, represents the final query solving this task.

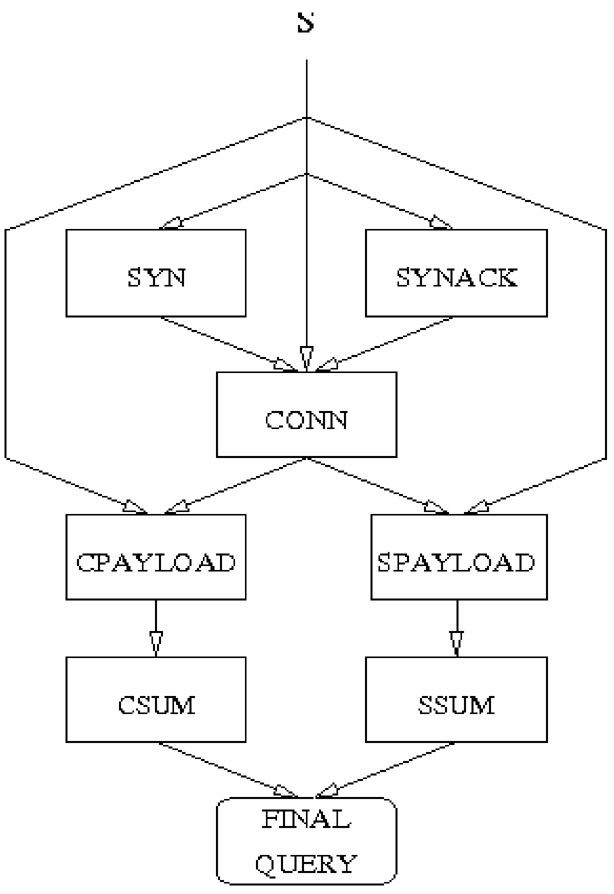


Figure 17. Data Flow for Task 4

The join in Conn is unblocked using a windowing operator. The size of this window decides the time available for connections to be established. For each step in the connection establishment, retransmissions may be experienced due to TCP congestion control mechanisms (Tanenbaum 2003). A packet is retransmitted when it is not acknowledged within some retransmission timeout (RTO) interval (Postel 1981b). Accurate dynamic determination of an appropriate RTO is essential to TCP performance. RTO is determined by estimating the mean and variance of the measured round-trip time (RTT) i.e. the time interval between sending a packet and receiving an acknowledgment for it (Jacobson et al. 1992). In Linux, `tcp_syn_retries` (found in /proc/sys/net/ipv4/) holds the number of times initial SYNs are retransmitted. The default value of this variable is five, which corresponds to approximately 180 seconds. For SYN/ACKs the default value is also five, corresponding to another 180 seconds (Postel 1981b). Hence, as a worst-case scenario, it may take as much as 360 seconds before the client receives the SYN/ACK message. The third step in the connection establishment may also experience retransmissions. The maximum number of retransmissions is in Linux found in `tcp_retries2` (found in /proc/sys/net/ipv4/). The default value is 15, which corresponds to a duration of approximately 13 to 30 minutes, depending on the retransmission timeout (Postel 1981b). Implementations of retransmission numbers may vary among different operating systems. In addition, RTOs are calculated dynamically. Consequently, it is difficult to give a worst-case estimate of how much time is required to establish connections. Hence, it is difficult to decide the window size in Conn.

In addition to deciding the time available for a connection to establish, the window size in Conn furthermore decides how long a connection stays alive. Figure 17 illustrates that Conn is joined with S in order to calculate the load for each of the connection ends. CPayload selects, for each connection, all packets sent from the client, while SPayload selects, for each connection, all packets sent from the server. It is no longer possible to identify packets belonging to a connection when that connection slides out of the window in Conn.

Depending on which application requesting a connection, connection duration may vary significantly e.g. loading a Web page versus transferring a large file. This leads to another problem with Conn's window size. For example, if we use a large window the load belonging to long-lasting connections are recognised for a longer period. However, this may result in several short-lasting connection, using the same quadruple, being counted as one. To solve the problem we may include recognition of connection closing, and the `EXCEPT` operator may be used to subtract the closed connections from established connections to recognised connections still staying alive. However, recognising closed connections is more complicated than recognising established connections, since the closing procedure includes more states. To complicate matters further, connections may be closed without accomplishing the steps involved in proper TCP connection termination. Hence, for simplicity we decide to merely include identification of connection establishment in our solution.

In our analysis, the window size is not of crucial importance. Our concern is that the query is syntactically and semantically correct, and that it produces the result we may expect. Thus, we use a three-minute window in Conn when presenting the solution. Independent of this window size, there are several limitations to our solution to Task 4. We present these limitations in the context of the three-minute window.

- All connections are considered to be alive for three minutes, regardless of when they actually are closed.
- Connections using more than three minutes to establish are not considered.
- If multiple connections are established over the same IP addresses, and port numbers, within the three minutes window, these connections are counted as one.
- The three-minute window starts based on the arrival time of the SYN packet in stream S, regardless of how much time is employed in order to accomplish the two following steps.

Our solution starts with the creation of the view Syn. This view contains packets where the SYN flag is set, and where the ACK flag is not set. These packets correspond to the first step in the handshake.

```
Syn (sourceIP, destIP, sourcePort, destPort, seqNum,
ackNum):

SELECT sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum

FROM S

WHERE SYN = "1" AND ACK = "0"
```

We also create a view Synack, where packets corresponding to the second step are selected. In these packets, both the `SYN` flag and the `ACK` flag are set.

```
Synack (sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum):
```

```
SELECT sourceIP, destIP, sourcePort, destPort,
seqNum, ackNum

FROM S

WHERE SYN = "1" AND ACK = "1"
```

The next step is to join Syn, Synack, and S, in the view Conn. This view contains connections where all steps in the handshake are accomplished. We use the *distinct* operator to exclude duplicates that would appear if packets in the connection establishment were retransmitted.

```
Conn (sourceIP, destIP, sourcePort, destPort):

SELECT DISTINCT SYN.sourceIP, SYN.destIP,
SYN.sourcePort, SYN.destPort

FROM Syn [RANGE 3 MINUTES], Synack [RANGE 3
MINUTES], S [RANGE 3 MINUTES]

WHERE Syn.sourceIP = Synack.destIP AND Syn.sourceIP
= S.sourceIP AND Syn.destIP = Synack.sourceIP AND
Syn.destIP = S.destIP AND Syn.sourcePort =
Synack.destPort AND Syn.sourcePort = S.sourcePort
AND Syn.destPort = Synack.sourcePort AND
Syn.destPort = S.destPort AND Syn.seqNum + 1 =
S.seqNum AND Synack.seqNum + 1 = S.ackNum
```

Notice that Conn is an intermediate relation. The content of Conn is not streamed as the window slides. It contains the IP addresses and port numbers from the client's side of an established connection. The next step is to join Conn with S to extract all packets in S, belonging to established connections in Conn. Since the solution proposed by Plagemann et al. (2004) only consider data bytes, a similar approach is taken here. To calculate the number of data bytes, we subtract header bytes from total number of bytes. Since IP- and TCP- header length are given in 32-bits words, we first multiply these header lengths by four to get header length in bytes. We use a one-minute window to unblock S. Consequently, we get packets received the last minute belonging to connections with a lifetime of three minutes. Two views are created, one for packets belonging to the client, and one for packets belonging to the server, respectively. Note that we in Spayload select the attributes in a different order. This is to represent the server's side by the client's representation.

```
Cpayload (sourceIP, sourcePort, destIP, destPort,
dataBytes):

SELECT S.sourceIP, S.sourcePort, S.destIP,
S.destPort, S.totalLength - ((S.ip_headerLength +
S.tcp_headerLength) * 4)

FROM Conn, S [RANGE 1 MINUTE]

WHERE Conn.sourceIP = S.sourceIP AND Conn.destIP =
S.destIP AND Conn.sourcePort = S.sourcePort AND
Conn.destPort = S.destPort



Spayload (sourceIP, sourcePort, destIP, destPort,
dataBytes):

SELECT S.destIP, S.destPort, S.sourceIP,
S.sourcePort, S.totalLength - ((S.ip_headerLength +
S.tcp_headerLength) * 4)

FROM Conn, S [RANGE 1 MINUTE]

WHERE Conn.sourceIP = S.destIP AND Conn.destIP =
S.sourceIP AND Conn.sourcePort = S.destPort AND
Conn.destPort = S.sourcePort
```

We use two more views to summarise the bytes belonging to each side of the connection, because STREAM does not support aggregations over arithmetic expressions.. CSUM represents the client's side, while SSUM represents the server's side.

```
CSUM (sourceIP, destIP, sourcePort, destPort,
dataBytes):

SELECT sourceIP, destIP, sourcePort, destPort,
SUM(dataBytes)

FROM Ipayload

GROUP BY sourceIP, destIP, sourcePort, destPort



SSUM (sourceIP, destIP, sourcePort, destPort,
dataBytes):

SELECT sourceIP, destIP, sourcePort, destPort,
SUM(dataBytes)

FROM Spayload

GROUP BY sourceIP, destIP, sourcePort, destPort
```

Finally, these views are joined to find how many bytes that have been exchanged on each connection during the last minute. As the window slides every second, we are interested in the sum of data bytes for all connections present in the window at that time instant, and not only the connections that include updates. Consequently, we use *Rstream* instead of *Istream* in the final query.

```
RSTREAM(

SELECT C.sourceIP, C.destIP, C.sourcePort,
C.destPort, C.dataBytes + S.dataBytes

FROM CSUM AS C, SSUM AS S

WHERE I.sourceIP = S.sourceIP AND I.destIP =
S.destIP AND I.sourcePort = S.sourcePort AND
I.destPort = S.destPort)
```

5.2.5 How many bytes are exchanged over the different connections during each week?

This task is rather similar to the previous one, except that it implies a window size of one week rather than one minute. This tasks is probably more suitable for statistics than for monitorin, because of the large window size. Hence, it may be solved better offline in a DBMS, than online in a DSMS. However, the task raises interesting challenges within the DSMS technology. As we mentioned in 5.2.4, the TelegraphCQ prototype used by Plagemann et al. (2004) did not support sub-queries. Another deficiency with that prototype was revealed when trying to solve this task. The window size must be smaller than one week, because all incoming data is kept in main memory until the entire window has been computed. With sliding window, each packet would contribute several times to intermediate results when the inter arrival time of packets is smaller than the window size. It is necessary to remove this redundant information to compute a correct result. To calculate absolute statistics, tumbling windows are needed (Plagemann et al. 2004). Tumbling windows were neither supported in the TelegraphCQ prototype, nor is it supported within STREAM. Consequently, this task is not possible to solve online with continuous queries in STREAM. In the latest TelegraphCQ prototype, there is support for both sub-queries and tumbling windows.

5.2.6 How much load on the university backbone have each department used within the past five minutes?

To solve this task, stream S must contain all packets from the university backbone. A stored relation, containing IP addresses belonging to departments, is joined with S. The content of this relation depends on the distribution of IP addresses in the university backbone. One subnet may contain several departments, with no ordered distribution of addresses. Alternatively, all addresses belonging to one department are found in a given range e.g. with each department having its own subnet. The University of Oslo's backbone has several departments on each subnet, with no ordered distribution of addresses within a subnet. Consequently, the stored relation must contain all IP addresses used by each department including the name of the department. This solution is similar to the one in (Plagemann et al. 2004). First, we define the relation:

```
REGISTER TABLE Departments (name char(30), ip_addr
char(16))
```

This relation is joined with S, based on equality of IP addresses. For each department, we summarise the number of bytes. Similar to Section 5.2.4, we are interested in data bytes. Consequently, header bytes are multiplied by four and subtracted from the total number of bytes. However, the STREAM prototype does not support aggregation over arithmetic expressions. Consequently, we first create a view, projecting source IP addresses and the arithmetic expression calculating data bytes. Finally, we join this view with the Departments relation in order to summarise bytes belonging to each department. Each second, as the window slides, the output should contain the number of bytes for all departments having traffic the past five minutes. Consequently, we encapsulate the query with `Rstream`.

```
Payload(sourceIP, dataBytes):

SELECT sourceIP, totalLength - ((ip_headerLength +
tcp_headerLength) * 4)
```

```
FROM S


RSTREAM(

SELECT D.name, SUM(S.dataBytes)

FROM Payload [RANGE 5 MINUTES] AS P, Departments AS
D

WHERE D.ip_addr = P.sourceIP

GROUP BY D.name)
```

5.2.7 What is the load on the network measured in connections per minute?

First, we recognise connections using the same approach as proposed in 5.2.4, where a connection is recognised through the views Syn, Synack, and Conn. We now count the connections per minute to solve the task. However, this is not as straightforward as it may initially appear. It is not possible to query Conn using windows, because it is an intermediate relation. In addition, if we count connections from Conn, we get the number of connections per three minutes. Thus, it may be tempting to use one of the relation-to-stream operators over Conn, and then query this stream using a one-minute window. However, such a query would not produce a correct result. As an example, consider a stream that only contains one connection. If we use `Rstream`, the connection is present in the stream every second. When counting connections from this stream over a one-minute window, the result would be 60. An obvious solution to this problem would be to use the `DISTINCT` operator, as in `COUNT(DISTINCT *)`. However, this solution is not supported within STREAM. If we use `Istream`, a connection is only present in the stream at the time when it is registered in Conn. When counting connections from this stream over a one-minute window, the result would be one. This is a correct result, but the problem is that this result would only be produced for one minute, because this solution would reduce a connection's lifetime from three minutes to one minute. In the solution we present, we do not stream the content of Conn. Instead, we unblock S using a one-minute window, and join S with Conn. The content of this view, which we call ConnMinute, is connections with a lifetime of three minutes that have exchanged packets during the last minute. It may be regarded as a weak point that the connections must transmit data during the last minute to contribute to the count. Nevertheless, a strong point may be that connections with an actual lifetime less than one minute only contribute to the count for one minute.

```
ConnMinute (sourcePort, destPort, sourceIP, destIP):

SELECT DISTINCT S.sourcePort, S.destPort,
S.sourceIP, S.destIP

FROM Conn, S [Range 1 minutes]

WHERE Conn.sourceIP = S.sourceIP AND Conn.destIP =
S.destIP AND Conn.sourcePort = S.sourcePort AND
Conn.destPort = S.destPort;
```

The final query counts the number of connections present in ConnMinute. We use `Rstream` to get the connections per minute each second.

```
RSTREAM(

SELECT COUNT(*)

FROM ConnMinute)
```

5.2.8 How often are HTTP and FTP ports contacted?

HTTP and FTP are application layer protocols and they both use TCP as transport protocol. FTP is divided into two modes: *active* and *passive*. We do not discuss the details of the FTP protocol and the difference of active FTP and passive FTP. However, in both modes a client initiates the FTP session by contacting the FTP port at a server. We define “contacting” as an attempt to open a connection i.e. the SYN bit is set and the ACK bit is not set. HTTP connections are initiated through port number 80, while FTP connections are initiated through port number 21. We define “how often” as the number of packets contacting HTTP and FTP ports during the past second. The following query should solve this task.

```
RSTREAM(

SELECT COUNT(*)
FROM S [RANGE 1 SECOND]

WHERE (destPort = 80 OR destPort = 21) AND SYN = "1"
AND ACK = "0"

GROUP BY destPort)
```

However, the current STREAM prototype does not support the `OR` operator. Consequently, another approach is necessary. We start by creating one view for all packets contacting http ports, and one view for packets contacting ftp ports.

```
HTTP (destPort):

SELECT destPort

FROM S

WHERE destPort = 80 AND SYN = "1" AND ACK = "0"


FTP (destPort):

SELECT destPort

FROM S

WHERE destPort = 21 AND SYN = "1" AND ACK = "0"
```

Next, we create another view, Contacts, where we calculate the union of the HTTP and FTP views.

```
Contacts (destPort)

HTTP UNION FTP
```

In the final query, we count the number of occurrences of each port number over a one-second window.

```
RSTREAM(

SELECT destPort, COUNT(*)

FROM Contacts [RANGE 1 SECOND]

GROUP BY destPort)
```

5.2.9 During the past minute, which connection contains most packets and how many packets does it contain?

Yet again we use the same approach as proposed in 5.2.4, where a connection is recognised through the views Syn, Synack, and Conn. Next, we create two views where we collect all packets sent during the past minute that belong to the different connections in Conn. Cpackets collects all packets sent from the client, while Spackets collects all packets sent from the server. Notice that we have turned around the order of the connection attributes to get a port number as the first attribute. The reason for this is that the count operator only manages to count tuples with an attribute of data type integer as the first attribute. When projecting attributes in Spackets, we also turn around the order of port numbers and IP addresses. This is to represent the server side of the connection in a similar manner as the client side.

```
Cpackets (sourcePort, destPort, sourceIP, destIP):

SELECT S.sourcePort, S.destPort, S.sourceIP,
S.destIP

FROM Conn, S [Range 1 minutes]

WHERE Conn.sourceIP = S.sourceIP AND Conn.destIP =
S.destIP AND Conn.sourcePort = S.sourcePort AND
Conn.destPort = S.destPort
```

```
Spackets (sourcePort, destPort, sourceIP, destIP):

SELECT S.destPort, S.sourcePort, S.destIP,
S.sourceIP

FROM Conn, S [Range 1 minutes]

WHERE Conn.sourceIP = S.destIP AND Conn.destIP =
S.sourceIP AND Conn.sourcePort = S.destPort AND
Conn.destPort = S.sourcePort
```

In the two previous views, it should be possible to group by connection and count the number of packets belonging to each group. However, STREAM does not accept this solution, and we are unable to explain why. The only error message we receive is “Error: Unknown error: -1.” Consequently, we create two more views, CC and SC.

These views query Cpackets and Spackets, respectively, in order to count the number of packets at each side of the connections.

```
CC (sourceIP, destIP, sourcePort, destPort,
pkCount):

SELECT sourceIP, destIP, sourcePort, destPort,
COUNT(*)

FROM Cpackets

GROUP BY sourceIP, destIP, sourcePort, destPort



SC (sourceIP, destIP, sourcePort, destPort,
pkCount):

SELECT sourceIP, destIP, sourcePort, destPort,
COUNT(*)

FROM Spackets

GROUP BY sourceIP, destIP, sourcePort, destPort
```

Next, we create another view, Connpackets, where we join CC and SC to summarise their count attributes. The two previous views do not contain any duplicates, because they include the `GROUP BY` clause. Thus, the join in Connpackets is a one-to-one join. Consequently, there are no duplicates in Connpackets.

```
Connpackets (sourceIP, destIP, sourcePort, destPort,
packetCount):

SELECT CC.sourceIP, CC.destIP, CC.sourcePort,
CC.destPort, CC.pkCount + SC.pkCount

FROM CC, SC

WHERE CC.sourceIP = SC.sourceIP AND CC.destIP =
SC.destIP AND CC.sourcePort = SC.sourcePort AND
CC.destPort = SC.destPort
```

In the next view, we query Connpackets to find the maximum number of packets during the last minute. We name the view Maxpackets.

```
MaxPackets (maxPacket integer):

SELECT MAX(packetCount)

FROM Connpackets
```

Next, we join Maxpackets with Connpackets based on equality of the number of packets. We select those connections having their total number of packets equal to the maximum number of packets. If several connections have equally many packets as the maximum, all these connections are selected.

```
RSTREAM(

SELECT C.sourceIP, C.destIP, C.sourcePort,
C.destPort, M.maxPacket

FROM Connpackets as C, MaxPackets as M

WHERE C.packetCount = M.maxPacket)
```

5.2.10 How long does a connection last?

The time at which a connection is established, and the time at which the connection is terminated, must be known in order to calculate the connection duration. Thus, connection duration may be defined as the difference in time between these two timestamps. Unfortunately, timestamps are not included among the attributes in S, nor does STREAM add any attribute, containing implicit timestamp, when tuples arrive in the system. A timestamp is added, but this is a timestamp used internally by the system, and not an attribute that may be queried. Consequently, queries discovering connection duration is not possible within the current STREAM prototype. However, if timestamps were included in S, we still have to recognise both connection establishment and connection termination. In 5.2.4, we discussed the complexity of connection recognition and termination.

5.2.11 For each pair of source and destination IP addresses, how many percent of the total load has it occupied during the past five minutes?

We still consider input stream S, representing data from the IP and IP layers, though we in this task only consider data from the IP layer. We define “load” to include the IP header i.e. we apply the total length field in the IP header. In our solution, we start by creating two views, one finding the total load of the network, and one finding the load for each pair of IP addresses, respectively.

```
TotalLoad (TLoad integer):
RSTREAM(
SELECT SUM(totalLength)
FROM S [Range 5 minutes])

FlowLoad (sourceIP, destIP, FLoad):
RSTREAM(
SELECT sourceIP, destIP, SUM(totalLength)
FROM S [Range 5 minutes]
GROUP BY sourceIP, destIP)
```

The final query joins the two previous views in order to calculate how many percent of the total load each pair of source and destination IP addresses have occupied during the past five minutes. We use the window size `NOW` to unblock both streams. This is equal to `RANGE 1 SECOND`, because windows in STREAM always slide each second.

```
RSTREAM(
SELECT F.sourceIP, F.destIP, (F.FLoad*100)/T.TLoad
FROM FlowLoad [NOW] AS F, TotalLoad [NOW] AS T)
```

5.2.12 Identify TCP SYN packets for which a SYN/ACK was sent, but no ACK was received within a specified bound of two minutes on the TCP handshake completion latency.

This task is fetched from the STREAM Query Repository (Anonymous 2002). The solution proposed there, assumes implementation of user defined methods, subqueries in the `WHERE` clause, the `EXISTS` operator, and negation e.g. `NOT EXISTS`. These features are not implemented in STREAM-0.6.0. Hence, we make another approach to solve this task. We start by creating two views Syn and Synack, which recognise the first two steps in the three-way handshake in TCP connection establishment. These views are similar to the ones we described in 5.2.4. Next, we create a third view, Ack, by joining Syn and Synack. Both streams are unblocked with a 120-seconds window. The tuples in ack represent the client's side of connections that have fulfilled the first two steps in the three-way-handshake. We use the relation-to-stream operator `Dstream` to stream tuples as they are deleted from the derived relation. The streamed tuples represent SYNs and SYNACKs that are two minutes old.

```
Ack (sourceIP char(16), destIP char(16), sourcePort
integer, destPort integer, seqNum integer, ackCheck
integer):

DSTREAM(

SELECT DISTINCT Syn.sourceIP, Syn.destIP,
Syn.sourcePort, Syn.destPort, Syn.seqNum,
Synack.seqNum from Syn [Range 120 seconds], Synack
[Range 120 seconds]

WHERE Syn.sourceIP = Synack.destIP and Syn.destIP =
Synack.sourceIP and Syn.sourcePort = Synack.destPort
and Syn.destPort = Synack.sourcePort and Syn.seqNum
+ 1 = Synack.ackNum)
```

Next, we use a similar approach as in 5.2.4 to create the view Conn, where all established connections are collected. However, in this solution we join the view Ack with S. Similar to section 5.2.4, we search S for packets belonging to the third step in the TCP connection establishment. Note that we use a window size of 121 seconds to unblock S in Conn. The reason for this is that we use `Dstream` and a window size of 120 seconds in Ack. This means that Ack produces results *after* 120 seconds have passed. At time t, Ack produces tuples with timestamp t, while S [Range n seconds] produces tuples with timestamps in the range $t - n$ to $t - 1$. As described in Section 4.3.2, STREAM requires that all queues enforce non-decreasing timestamps. The join does not know that it has received all tuples from S with timestamp $t - 1$ before it sees a tuple with timestamp t. This happens at time $t + 1$. However, Ack streams tuples with timestamp $t + 1$ at time $t + 1$. Thus, the join operator can never join tuples from the two streams if we use a window size of 120 seconds to unblock S.

```
Conn (sourceIP char(16), destIP char(16), sourcePort
integer, destPort integer, seqNum integer, ackNum
integer):

Istream (

SELECT DISTINCT Ack.sourceIP, Ack.destIP,
Ack.sourcePort, Ack.destPort, Ack.seqNum,
Ack.ackCheck

FROM Ack [Now], S [Range 121 seconds]

WHERE Ack.sourceIP = S.sourceIP and Ack.destIP =
S.destIP and Ack.sourcePort = S.sourcePort and
Ack.destPort = S.destPort and Ack.seqNum + 1 =
S.seqNum and Ack.ackCheck + 1 = S.ackNum)
```

In Ack, we have attributes from SYN packets where the corresponding SYNACK packet is sent. In Conn, we have established connections. Thus, to find SYN packets, where no acknowledgement was received within a specified bound of two minutes on the TCP handshake completion latency, we calculate the difference between Ack and Conn. In the final query, we use the `EXCEPT` operator to calculate this difference.

```
ISTREAM (Ack EXCEPT Conn)
```

The solution in the STREAM Query Repository (Anonymous 2002) projects all attributes belonging to a packet. To accomplish this, we could create a view of the final query. Then we would join this view with S to identify the SYN packets and extract all the attributes. However, STREAM does not support this solution, because it conflicts with a constant, named `MAX_INSTR`, in the source code. We may easily change this constant and recompile STREAM. However, we do not change the source code, because we treat STREAM as a black box.

6. System Implementation

In this section, we implement the experiment setup for the performance evaluation. We first add functionality to STREAM to make it possible to receive data from online sources, because this is not implemented in the current prototype. In addition, we implement a packet capturing system to capture packets from the network. Next, we present the experiment architecture along with descriptions of computers, programs, and system monitors used within the experiment. We also perform some preliminary tests to investigate the capacity and accuracy of our system, the overhead introduced by other processes than STREAM, and the packet sizes produced by the traffic generator.

Recall from Section 2.2.1 that traffic measurements are divided into two different techniques; *passive* and *active* measurements. The experiment setup we implement simply observes and records the traffic as it passes by. Consequently, we use passive measurement technique in our experiments.

6.1 Live Data Source

STREAM provides three classes: Server, TableSource, and QueryOutput, which constitute its external interface. The main method in TableSource is `getNext()`, which the server uses to pull the next tuple of the stream or relation whenever it desires. In the generic client, gen_client, there are two classes implementing the TableSource interface; RelationSource and StreamSource. As the names suggest, their `getNext()` methods read data belonging to relations and streams, respectively. However, both methods are implemented to only read data from stored files, and do not support reading data from live sources. Consequently, we implement a third class, SocketSource, to allow reading of data from live streams. As this name suggests, the `getNext()` method reads data from a socket. To avoid packet loss due to a full socket buffer, we choose to receive data over a TCP connection, which guarantees correct and orderly delivery of data across the network. Moreover, compared to other sources of data e.g. a Linux pipe, a TCP connection may receive data from a distant source. Although we implement SocketSource to allow only one TCP connection, it can easily be expanded to read data from many streams, each arriving over one TCP connection. This allows distributed data collection, and introduces new applications such as comparing traffic from distant locations in the network. As described in Section 4.4.3, a source is required when defining a table. When the source is a file, the location of the file is given. To read data from the socket, the word "Socket" is given as source. As described in Section 4.4.4, the first line in the table source file represents data types according to the table's schema. Thus, such a schema-describing line is sent to STREAM prior to sending any tuples.

In Section 5, we designed queries to solve network traffic monitoring tasks. The queries are posed over an input stream where a tuple consists of TCP and IP header field values. Thus, in addition to implementing streaming of data to STREAM over a TCP connection, we also implement a system that captures packets from the NIC. In turn, the packet headers are sent over the TCP connection to STREAM for further processing.

6.2 Packet Capturing

Many packet-capturing tools are available as freeware. One well-known packet-capturing tool is Tcpdump. Tcpdump captures packets and writes their content to screen according to a rich variety of options. Tcpdump does not send this output over a TCP connection, nor does it produce packet header fields as tuples of comma-separated values (CSV), which is the representation used in STREAM. There are several solutions to these problems:

1. We may use Tcpdump as released, writing packet header fields in a hexadecimal notion (optional) to screen. We then implement a filter that reads from standard in, and transforms the hexadecimal notion into CSV tuples. The filter then sends these tuples to STREAM. A Linux pipe is used between Tcpdump and the filter.

2. We may add extra functionality to the current Tcpdump implementation. Tcpdump then captures packets, transforms header fields into CSV tuples, and sends these tuples to STREAM.

3. We may implement a program from scratch in order to capture packets from the NIC, extract header fields from the raw data, transform the header fields into CSV tuples, and send these tuples to STREAM.

The first solution introduces a problem with determinism. We are interested in measuring the performance of STREAM, and not how a Linux pipe performs. If the TCP buffer in STREAM gets full, TCP congestion control mechanisms slows down the rate at which the filter sends packets. In turn, this leads to more data in the pipe between Tcpdump and the filter. If the pipe consumes to much memory resources, data in this pipe is written to disk. This slows down reading and writing in the pipe. In turn, this may force the operating system to schedule more processing time to other processes than STREAM. Consequently, we prefer one of the two last solutions.

Tcpdump is a relatively large program with many functionalities that mostly are unnecessary for our purpose. Consequently, we decide to implement a packet capturing and filtering program that are dedicated to meet our requirements. We implement this program from scratch and make it Fyaf. In Fyaf, we also include some monitoring applicability that we use in our performance evaluation. Fyaf will be further described in Section 6.3.3.

6.3 Experiment setup

Prior to completing experiments used to measure STREAM's performance, we describe the architecture of the setup for these experiments. In addition, we describe the computers and the software that are included in the experiments.

6.3.1 Experiment architecture

In Section 5.2, we described several tasks that recognise TCP connections. We crate an experiment setup that may generate traffic over several TCP connections, because of these tasks. In addition, we want an experiment setup that is both as close to the real world as possible and that offers the ability to recreate the workload from execution to execution i.e. it is repeatable. The architecture of our experiment setup is illustrated in Figure 18 below. It consists of two computers, which are referred to as computer A and B. These computers are connected to one another via an Ethernet crossover cable. Up to *n* instances of the traffic generator TG run on each computer, one instance on A communicating over a TCP connection with one instance on B. TG allocate a unique socket number for each connection. Consequently, we have the possibility of creating distinct TCP connections based on the quadruple described in Section 5.2.4. The TG instances on A are the client sides of the connections, while the TG instances on B are the server sides. On computer B, packets are captured from the NIC, and delivered to Fyaf. Further, Fyaf transforms TCP/IP header values into CSV tuples and sends these tuples over a TCP connection to STREAM.

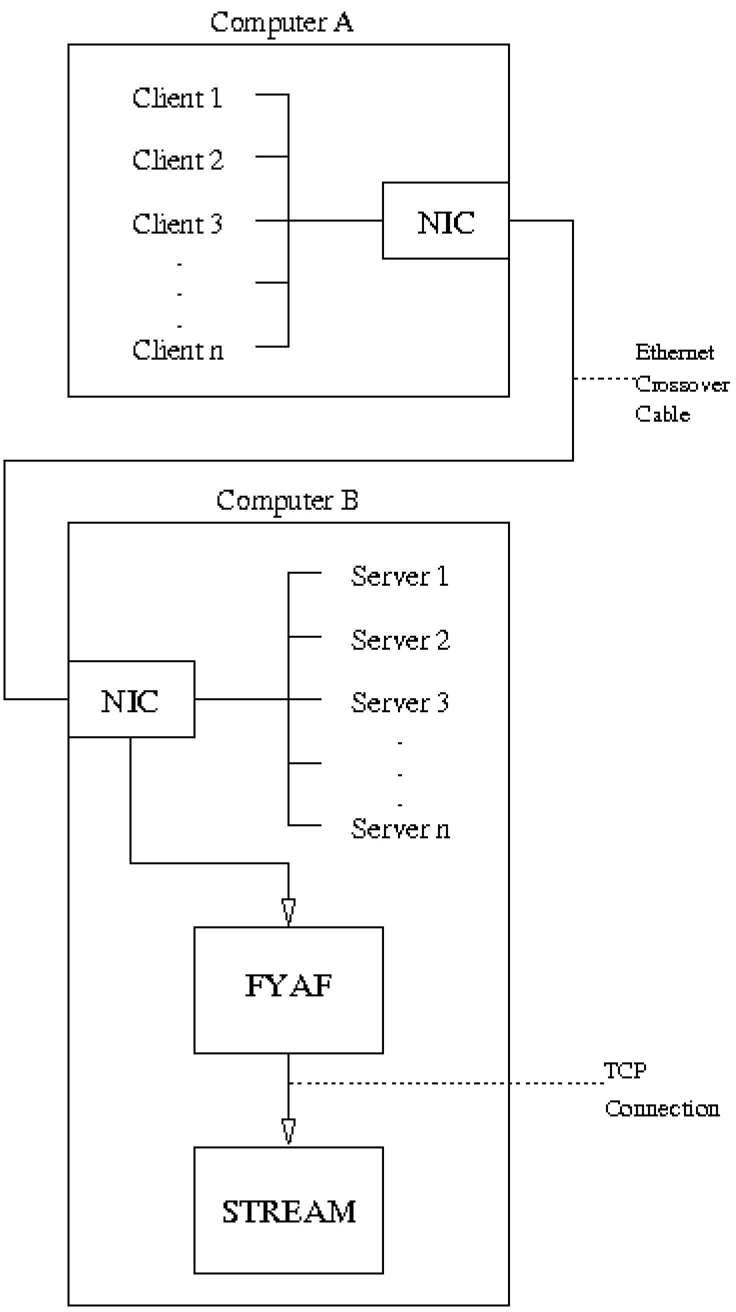


Figure 18. Architecture of experiment setup

6.3.2 Computers

As illustrated in Figure 18, we use for our experiments two different computers, computer A and B. These computers have equal specifications. They both run a SuSE Linux 9.2 with a 2.6 kernel, and have a memory size of 1.00 GB. The computers have two processors each, all of which are an Intel® Pentium® 4 CPU 3.00 GHz with a cache size of 1024 KB. Both computers have Gigabit Ethernet cards.

6.3.3 Programs

The experiment setups make use of several programs to generate traffic, capture packets, filter packets, and to automate the execution of our experiments. In this section, we give a short description of these programs.

TG

There are many traffic-generating tools available as freeware (e.g. Iperf and TG). When testing different tools we find that TG is easy to understand, and it has a simple user interface that is uncomplicated to use. Consequently, we choose TG as our traffic-generating tool.

TG is developed at SRI International, and is implemented to characterise the performance of packet-switched network communication protocols. The TG program generates and receives one-way packet traffic streams transmitted from the UNIX user level process between traffic source and traffic sink nodes in a network (McKenney et al. 2002). The traffic is described in terms of inter arrival times and packet lengths. TG allows packets to be sent from one client to one server. To send data over multiple connections, we run multiple instances of TG. When sending TCP packets, TG sends 12 bytes in the TCP option field, which results in a TCP header size of 32 bytes. The TG program records all notable events in a log file for post-test analysis. With the exception of the header descriptor, the TG log file is encoded in binary form to conserve storage space.

Fyaf

We implement Fyaf to use the Pcap library in order to capture packets from the NIC. This library is also used by Tcpdump. In addition to capturing packets, transforming them to CSV tuples, and sending them to STREAM, Fyaf also writes monitoring information to files:

- `slog.log` logs the number of packets received on the NIC and the number of packets dropped due to system resources. This information is provided by Pcap. In addition, we log the number of packets that is captured by Fyaf i.e. the number of packets received on the NIC that is not dropped.
- `f2.log` logs the traffic duration i.e. the number of seconds at which Fyaf has registered traffic.

Fyaf is mainly implemented by Jarle Søberg, and a thorough description of the program is provided by Søberg (2006).

Scripts

In order to automate the execution of the different experiments, we introduce some scripts that accomplish this task.

- Computer A
  - `experiment_server.pl` establishes a connection with `experiment_client.pl` on Computer B. This connection is used to exchange commands and synchronise the traffic generation. The script uses the commands received by `experiment_client.pl` to execute other scripts.
  - `change_template.pl` is executed by one of the commands received by `experiment_client.pl`. This script uses the different parameters sent from `experiment_client.pl` to calculate different values that are used to characterise the TG stream. In addition, it calls `create_clients.sh` with these values.
  - `create_clients.sh` creates files that describe the TG clients. These files contain information such as packet size.

    - `tg_client_run.pl` is executed by one of the commands received by `experiment_client.pl`. This script starts the number of TG clients that are used in the experiment.

- Computer B

    - `sscript.pl` contains information of the experiments we perform. Examples of such information are what experiments are to be executed, how many connections that should generate traffic, which network load to use, and how many times to perform the experiment.

    - `super_script_2.pl` is called by `sscript.pl`. It iterates over the number of executions. For each iteration, the script starts and stops STREAM, calls `experiment_client.pl`, and copies all necessary files to the folder for this iteration after it is finished.

    - `experiment_client.pl` is called by `super_script_2.pl`. This script starts Fyaf, TG's packet transfer, and the system monitors. In addition, it calls `create_servers.sh` and `tg_server_run.pl`.

    - `create_servers.sh` is called by `experiment_client.pl`. It creates files that describe the TG servers. These files contain information such as packet size.

    - `tg_server_run.pl` is called by `experiment_client.pl`. It starts the number of TG servers that are used in the experiment.

System Monitors

- `Top` allows us to view the process table in order of CPU or memory usage, by user, and at varying refresh rates. It is useful for viewing real-time process behaviour.

- `Sar` produces system utilisation reports based on the data collected by `Sadc`, which collects system utilisation data and writes it to a file for later analysis.

- `Vmstat` produces an overview of process, memory, swap, I/O, system, and CPU activity.

6.4 Preliminary Tests

The architecture of the experiment setup in Figure 18 illustrates that one instance of TG is running on computer B for each TCP connection between computer A and computer B. In addition, Fyaf, scripts, and system monitors are running on this computer. Prior to introducing the performance evaluation, we perform some preliminary tests on our experiment setup. The first test investigates whether TG produces the requested network load. The second test investigates whether TG produces packets with the requested packet size. The third test investigates the capacity of Fyaf. The fourth test investigates whether Fyaf and STREAM are accurate. Finally, the fifth test investigates whether the scripts and monitors introduce any significant consumption of system resources.

6.4.1 Network load

To be able to include network load as a factor in our performance evaluation, we perform a small test where we request TG to generate traffic and use Sar to see how well it has been done. Sar reports how many bytes are received on the NIC. In this test, we use the experiment setup described in Section 6.3. STREAM processes the query from Task 1 under varying network load. The network load that is registered on the NIC is portrayed in Figure 19 below. Each curve represents a requested network load, and illustrates the actual network load measured on B. Figure 19 shows that the network load is very much as we request up to 10 Mb/s, and that it produces 1 Mb/s or 2 Mb/s less when requesting 30 Mb/s and 50 Mb/s.

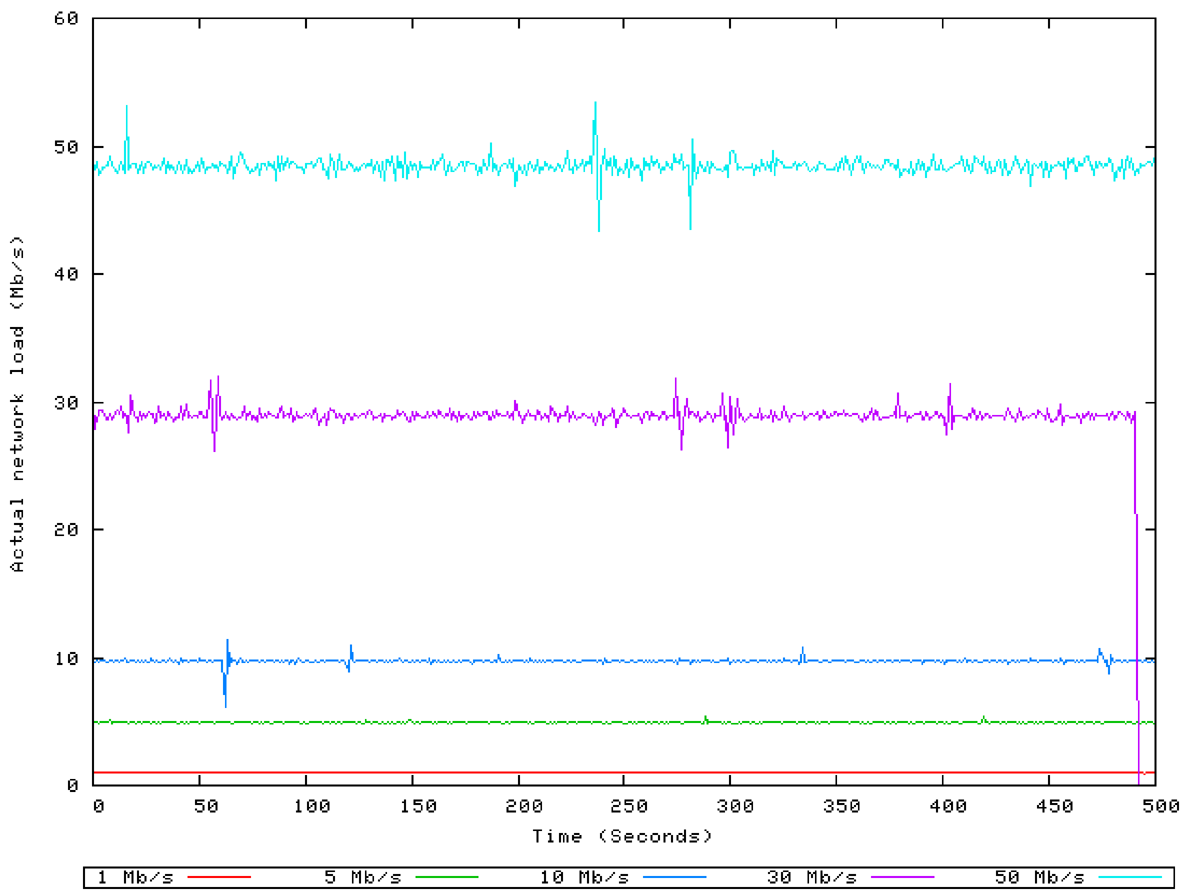


Figure 19. Actual network load produced by TG.

In our experiment setup, we capture packets on Computer B. On the NIC, there are packets sent from A to B, but also packets sent from B to A. Packets sent in both directions are captured. However, there are only acknowledgements sent from B to A, because the TG client generates the actual traffic. Some of our network-monitoring queries produce an answer that is composed by both the original packets and their acknowledgements, because they query the total load on the network, they. When we calculate the accuracy of the query answers, we use the requested network load as an expected answer. Consequently, we investigate how many percent of the total load that is composed by acknowledgments, because the network load used as factor only represent traffic in one direction. We perform this test over the same Sar files as we used to generate Figure 19. The result from this test is illustrated in Figure 20 below. We see that acknowledgements constitute approximately 6.3 % of the total load at 1 Mb/s, and that the percentage decrease as the network load increase.

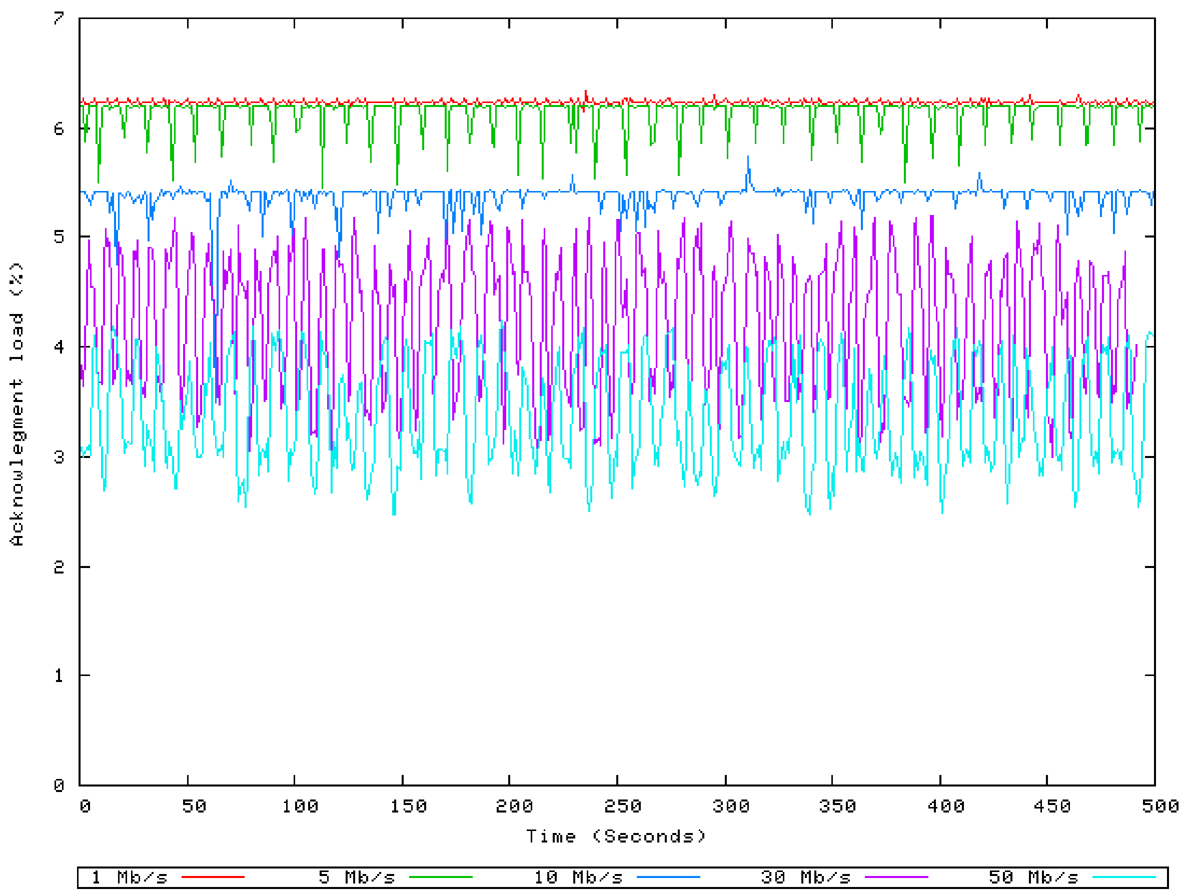


Figure 20. Percentage of total load introduced by acknowledgements

6.4.2 Packet Size

To confirm accuracy in queries counting packets we may use the network load and the packet size to calculate an expected number of packets per second. We instruct TG to send packets of a given size and at a given inter-arrival time. Together these parameters decide the network load. However, when investigating packets received on the NIC, we find that not all packets have the requested size. If we request small packets to be sent at a given network load, and large packets are sent instead, this may have great impact on the number of packets per second. Consequently, we perform a small test to find the average packet size at different network loads. We run TG and Fyaf, but instruct Fyaf to write the tuples to file. For each network load, we then insert the tuples into a PostgreSQL relation, which we query to obtain the average packet size. We use a small collection of network loads and only one run for each. The runs have a duration of 15 minutes, and a requested segment size of 576 bytes, which does not include TCP and IP headers. When including TCP and IP headers, the packet size should be 628 bytes. The average packet sizes of the packets sent by TG at different network loads are presented in Figure 21 below. The mechanisms attached to TCP include fragmenting segments in order to send as large packets as possible. This is to reduce the overhead on the network introduced by packet headers. Thus, TG probably sends segments of 576 bytes to the TCP layer, which handles these packets, as it desires. From Figure 21, we see that as the network load increase, the gap between the requested packet size and the average packet size also increases.

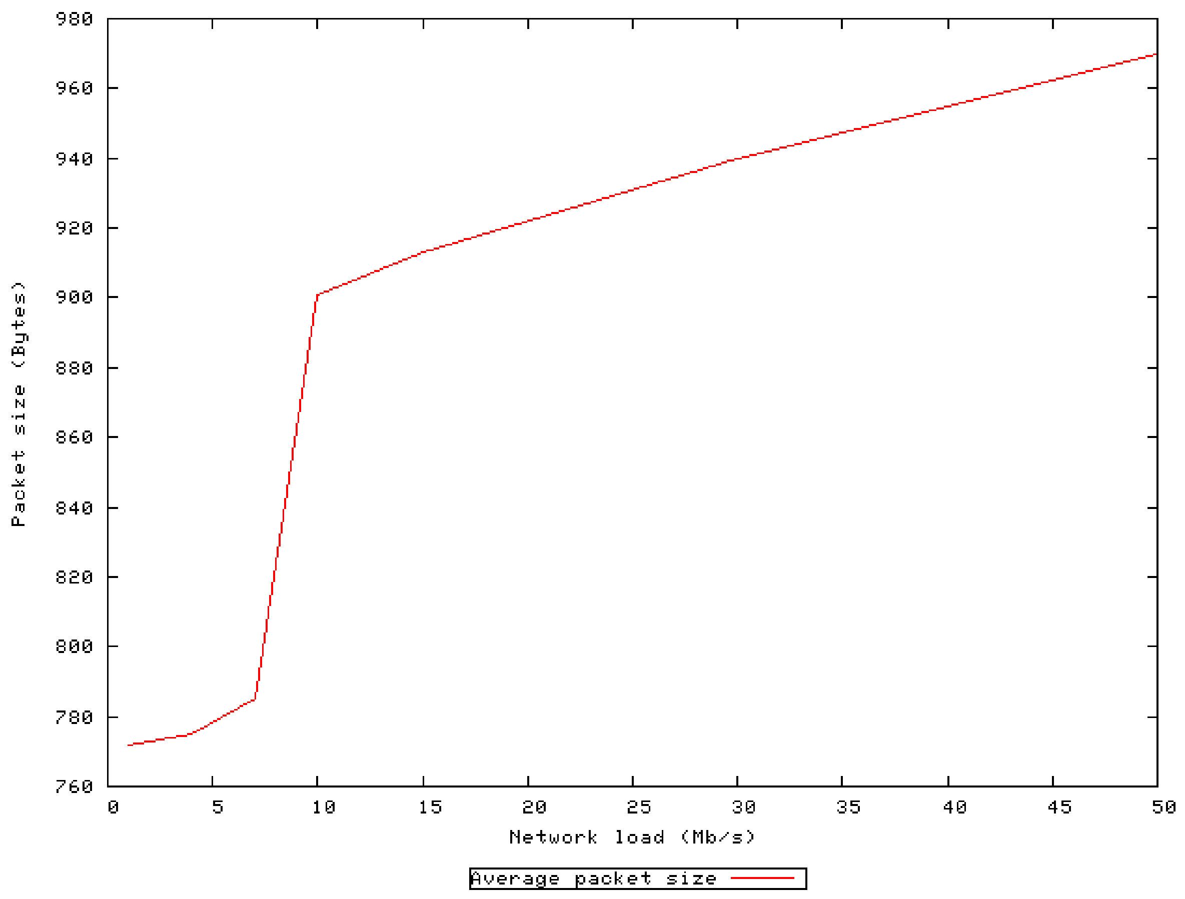


Figure 21. Average packet size produced by TG

6.4.3 Fyaf Capacity

In this test, we investigate the capacity of Fyaf to decide whether Fyaf is a bottleneck in the experiment setup described in Section 6.3. However, instead of sending packets from Fyaf to STREAM, we implement a sink that Fyaf connects and sends packets to. This sink only reads data from the TCP buffer. It does not process the data in any way. We instruct TG to generate traffic for five minutes, and we vary the network load at which data is sent from A to B. For each network load, we execute five runs and measure the average percentage of packets dropped due to system resources. The result of this test is presented in Figure 22(a) below. We see that less than 0.04 % of the packets are dropped for all network loads. For most network loads, less than 0.02 % of the packets are dropped. If we subtract 0.04 % from 50 Mb/s, this results in 49.98 Mb/s. To find whether Fyaf is a bottleneck, we run a test on the experiment setup from Section 6.3, which means that Fyaf sends packets to STREAM instead of the sink. STREAM processes the query from Task 1 over traffic with a duration of five minutes. For each network load, we repeat the run five times. The result from this test is presented in the Figure 22(b) below. The red curve in Figure 22(b) is the same as the curve in Figure 22(a). In Figure 22(b), we see that the packet dropping is close to zero when STREAM is not included. When we include STREAM, packet dropping increases more rapidly as network load increase, especially from 30 Mb/s. Thus, Fyaf is not a bottleneck in our experiment setup.

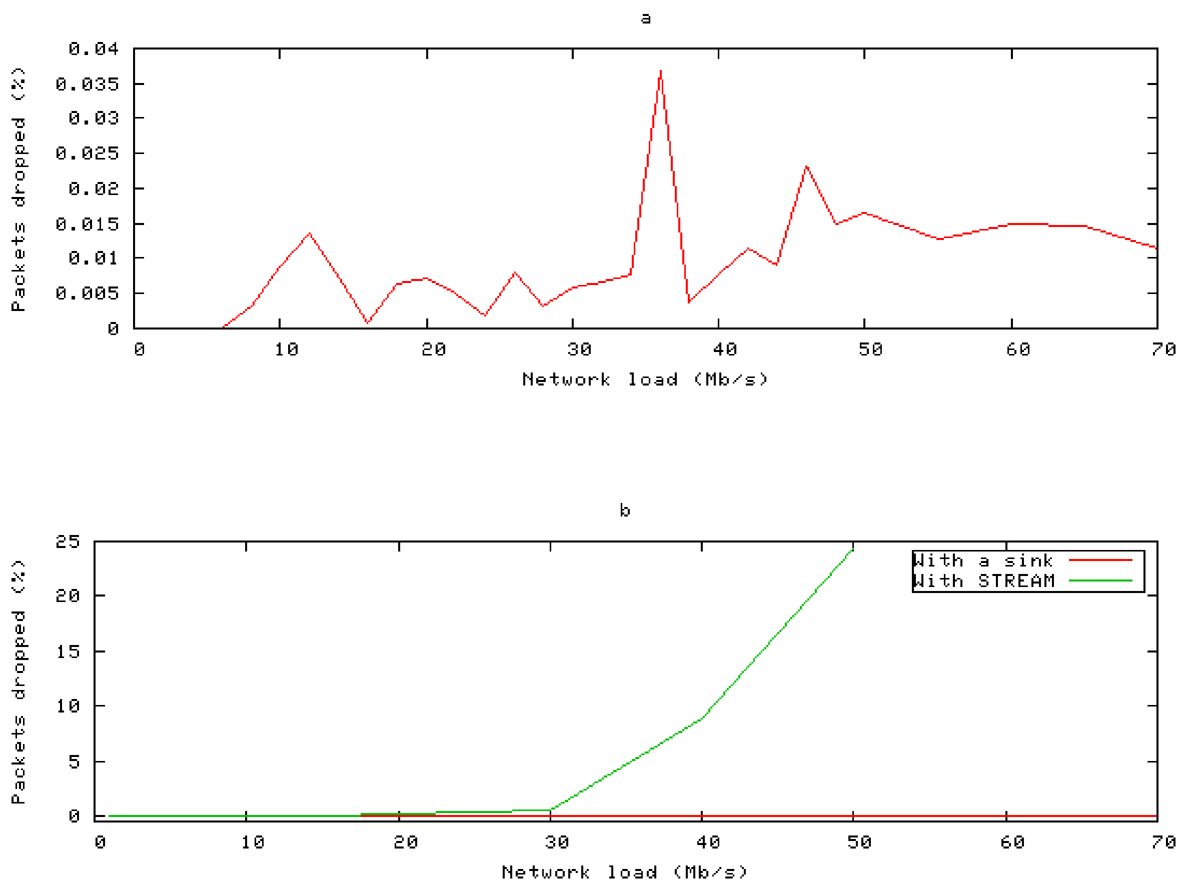


Figure 22. Packet drop rate in percent for Fyaf and a sink (a) and Fyaf and a sink compared to Fyaf and STREAM (b)

6.4.4 Accuracy of Fyaf and STREAM

As mentioned in Section 6.3.3, Fyaf reports the number of packets that are seen on the NIC, the number of packets that are dropped due to system resources, and the number of packets Fyaf captures. Since load shedding is not implemented in STREAM, the only statistics we have on packet dropping are the numbers reported by Fyaf. Consequently, we use the statistics produced by Fyaf when measuring STREAM's performance. Thus, prior to the measurement of performance, we evaluate the correctness of these statistics. We investigate whether the number of packets reported by Fyaf agree with the number of packets received by STREAM. They should correspond, because data is sent from Fyaf to STREAM over a TCP connection, which guarantees correct and orderly delivery of data across the network. Further, we investigate whether STREAM processes these data correctly by comparing the query results of two queries with the number of tuples received from Fyaf. We add new functionality to Fyaf to make these comparisons possible. We count the number of packets captured by Fyaf that are TCP/IP packets, and how many are not. If packets that are not TCP/IP packets appear in Fyaf, we print their content to file for further analysis. The number of packets that are TCP/IP packets are the number of packets that Fyaf sends to STREAM. We define two different queries for this test:

Query 1

```
SELECT *
FROM S
```

Query 2

```
SELECT COUNT(*)
FROM S
```

We execute the queries over streams with duration of 60 seconds and vary the network load at which data is sent from A to B. Furthermore, we define four requirements that the results from this test must fulfil in order for us to accept accuracy of the interaction between Fyaf and STREAM.

1. The number of packets received on the NIC minus the number of packets dropped must equal the number of packets captured by Fyaf.
2. The number of packets captured by Fyaf minus the number of packets sent to STREAM must equal the number of packets registered in Fyaf that are not TCP/IP packets.
3. The number of TCP/IP packets captured by Fyaf must equal the number of packets received by STREAM.
4. The number of packets received by STREAM must equal the number of packets in STREAM's query result.

To ensure that the fourth requirement is fulfilled we investigate the result files from the two queries. Regarding Query 1, we count the number of lines in the result file, whereas we in Query 2 use the last result in the result file. The reason for this is that STREAM writes each update of the intermediate relation to file. When counting, each tuple that arrives the count operator represents an update.

The results from our test indicate that all four requirements are fulfilled. In addition, the results reveal that not all packets received by Fyaf are TCP/IP packets, even though we instruct TG only to send data over TCP connections. When investigating these packets, we find that they are all Address Resolution Protocol (ARP) packets. An ARP packet typically looks like (in hexadecimal notion):

```
00118560ff0e001185172ca10806000108000604000100118517
2ca181f04341000000000000081f0433c
```

The number `0806`, in letter 25 to 28, identify it as an ARP packet. Further descriptions of ARP are given in (Plummer 1982). The ARP packets compose a part of the statistics provided by Fyaf, but not a part of the packets sent to STREAM. Since we use these statistics to evaluate the results of our experiments, we need to investigate the margin of error composed by the ARP packets. We calculate the margin of error by comparing the number of ARP packets to the total number of packets captured by Fyaf. Figure 23 below shows this rate in percent. The red line is hidden behind the green from 1 Mb/s to 5 Mb/s. We see that the margin of error is less than 0.025 % at a rate of 1 Mb/s, and that it approaches 0 % as the network load increases. Consequently, we consider this margin of error as insignificant.

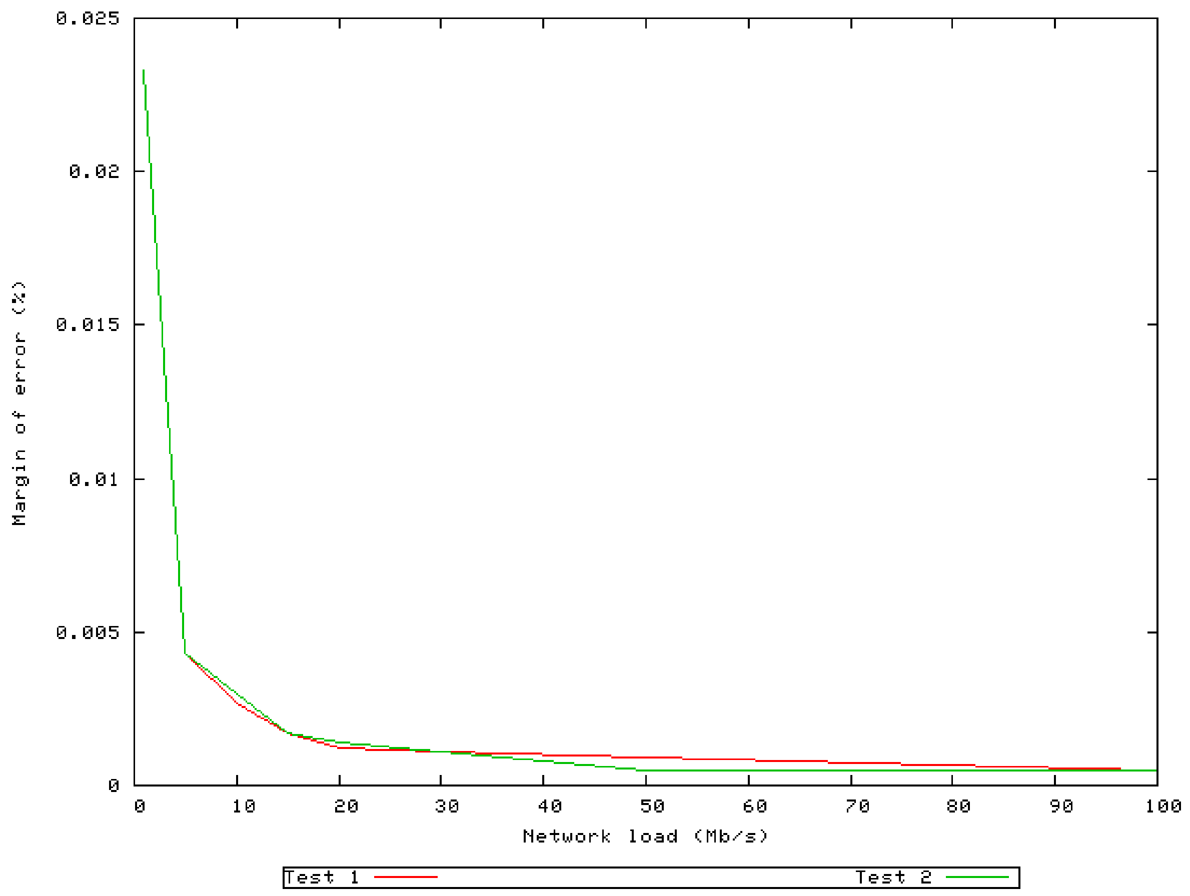


Figure 23. Margin of error at different network loads

6.4.5 System Resource Consumption

As we described in Section 6.3, we introduce many programs in our experiment setup. Some of these programs create several processes. In this section, we investigate whether these processes introduce any significant consumption of system resources. We use the online experiment setup to run the query from Task 3 and define the relation to consist of all 65536 port numbers. We run the test one time, and we produce traffic over one TCP connection at a rate of 5 Mb/s. In order to analyse system resource consumption, we use the file produced by Top. Top is instructed to report every second.

Figure 24 below presents CPU consumption introduced by other user processes than STREAM. Only three processes, Sadc, Tcp_server, and Top are present in the figure.

The reason for this is that Top reports CPU and memory consumption with a precision down to one-tenth percent. When a process consumes very little CPU resources, this is registered as 0.0 % by Top. Two of the curves in Figure 24 are not easily separated from each other. The Top process, which is represented by the red curve, consumes two percent or less, while Sadc, represented by the green curve, consumes one percent or less. The Tcp_server process periodically consumes up to 60 % of the CPU resources. About 120 seconds elapse between each peak. Between the peaks, Tcp_server consumes 0.0 %. We search the files produced by Top, Sar, and Vmstat without finding any reasons for these peaks. However, by investigating the same files, we find that STREAM's CPU consumption is not influenced by this process and its peaks. Consequently, CPU consumption introduced by other processes than STREAM is not significant.

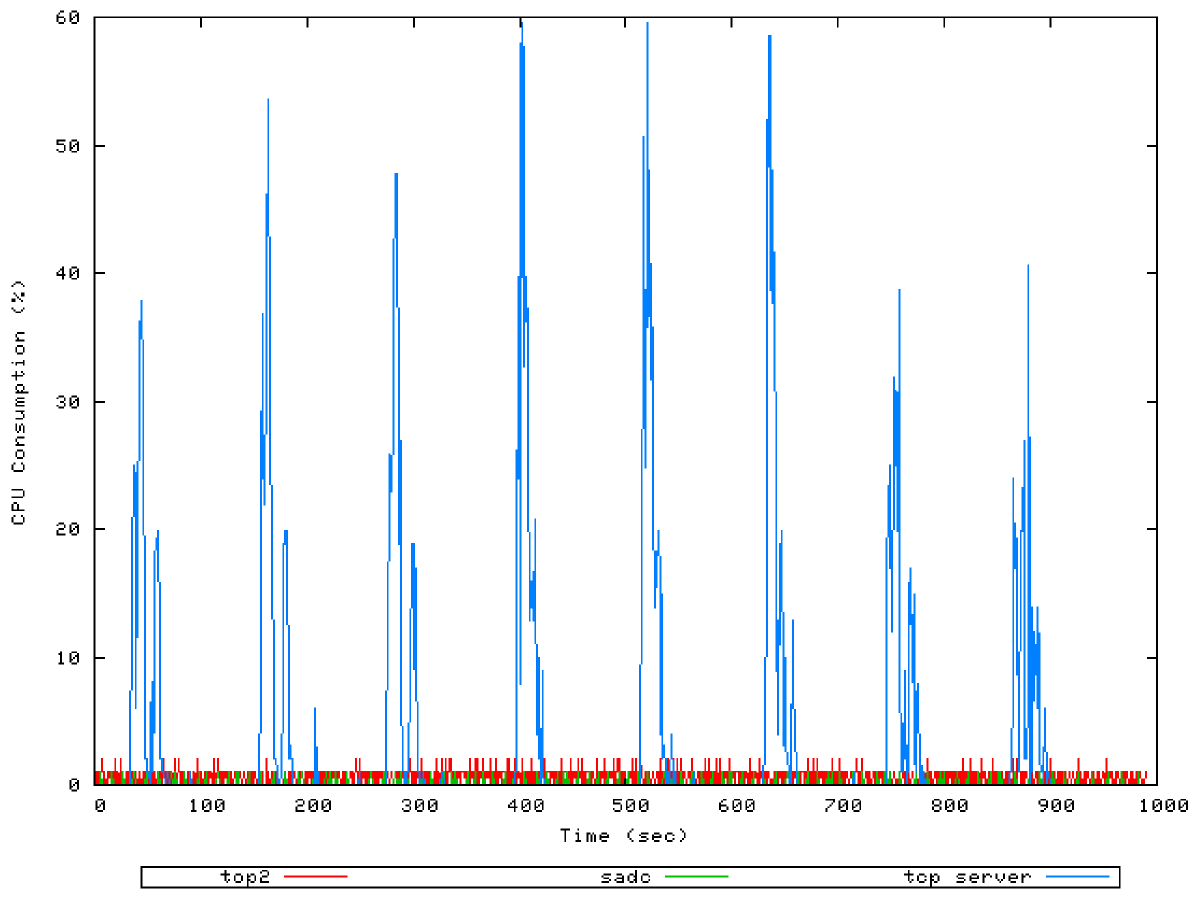


Figure 24. CPU consumption of other processes than STREAM

The memory consumptions, introduced by other processes than STREAM are presented in Table 4 below. We see that none of these processes consumes more than 0.3 %. The data files, suggests that the memory consumption is constant throughout the run. One of the Vmstat processes is registered with a memory consumption of 0.0 %. This is due to the Top precision. The total memory consumption introduced by these processes is 2.9 %. The fact that STREAM allocates memory statically, based on the configuration file, signify that STREAM consumes approximately 76.8 % of the total memory. Thus, the memory consumption introduced by other processes that STREAM is insignificant.

Process	Memory Consumption (%)	Process	Memory Consumption (%)
experiment_client1	0.3	fyaf	0.1
experiment_client2	0.3	sadc	0.1
experiment_client3	0.3	sar	0.1
experiment_client4	0.3	tcp_sever	0.1
sscript	0.2	top1	0.1
super_script1	0.2	top2	0.1
super_script2	0.2	vmstat1	0.1
tg_server_run1	0.2	vmstat2	0.1
tg_server_run2	0.2		

Table 4. Memory consumption of other processes than STREAM

7. Performance Evaluation

In this section, we present the performance evaluation. We start by defining evaluation technique, metrics, factors, and workload. Then, we design different experiments to use in the evaluation. To accomplish the performance evaluation, we use a similar approach as described by Jain (1991).

Jain (1991) divides performance evaluation into three techniques: *analytical modelling*, *simulation*, and *measurement*. He also states that analytical modelling and simulation may be used in situations where measurement is not possible. Moreover, we are interested in measuring the performance of the prototype under real, but controlled, network traffic. Consequently we choose *measurement* as our evaluation technique.

Recall that we evaluate STREAM's performance in the context of regarding it as a black box, which was defined in Section 1.2. We do not change the source code in order to implement additional functionality. However, we investigate the source code to better understand the internal mechanisms in STREAM.

7.1 Metrics

Metrics are the criteria in which to measure the performance. In general, the metrics are related to *responsiveness*, *productivity*, and *utilisation* (Jain 1991). Responsiveness is the time taken to perform a service, productivity is the rate at which the service is performed, and utilisation is the resources consumed while performing the service. A DSMS continuously produces results, because it processes a continuously arriving stream of data elements. Consequently, it continuously performs services and it is therefore not possible to measure responsiveness of a single service without adding monitoring capabilities to STREAM. We use *relative throughput* as a productivity metric, and *consumption* as a utilisation technique. In addition, we measure STREAM's *accuracy*.

- **Relative throughput**: We use relative throughput to measure how many percent of the offered network load that has been handled by Fyaf and STREAM. We use the statistics reported by Fyaf to calculate the relative throughput. Consequently, relative throughput is the number of packets captured and processed by Fyaf and STREAM relative to the total number of packets registered on the NIC, measured in percent. Recall from Section 6.4.3 and 6.4.4 that Fyaf introduces no overhead compared to STREAM and that the number of TCP/IP packets captured is equal to the number of tuples received in STREAM. Since STREAM may crash under heavy load, we only consider the relative throughput for those runs that are executed correctly. If there are no correct runs, relative throughput equals zero.

- **Consumption**: We use consumption as a measurement for the system resources occupied by STREAM. We measure consumption in terms of CPU usage. We do not measure memory consumption, because STREAM allocates memory statically based on the memory parameter in the configuration file. Thus, STREAM's memory consumption is constant.

- **Accuracy**: In addition to relative throughput and consumption, we also measure the accuracy of STREAM. To measure accuracy we investigate the query results produced by STREAM. We compare these results with the results we expect based on the given workload. Several factors make such an measurement difficult. For example, the network load produced by TG is neither constant over time, nor does it exactly match the requested network load, as described in Section 6.4.1. In addition, the packets sent by TG are of varying sizes, as indicated in Section 6.4.2. Thus, it is difficult to calculate the answer to expect. Nevertheless, analyses of the query answers should implicate whether the query answer is correct or not. A drop in relative throughput may have influence on the query result e.g. when the querying the network load. Consequently, we should only analyse accuracy on answers produced under a relative throughput close to 100 %. Since there are several factors of uncertainty in this matter, we accept a relative throughput down to 98 % when investigating the accuracy of the query results.

7.2 Factors

Parameters affect the system performance, and are divided into *system parameters* and *workload parameters* (Jain 1991). System parameters include both hardware and software parameters, which generally do not vary among the various executions of the system. User requests, which may vary from one execution to another, characterise workload parameters. Further, parameters may be divided into two parts; those that will be varied during the evaluation and those that will not. The parameters to be varied are called *factors* and their values are called *levels* (Jain 1991). The factors we use in our evaluation are the following.

- **Relation size**: There are several conventions used for describing the size of a relation *R*. In Section 3.2, we described the relation size as the number of blocks that are needed to hold all the tuples of *R*. This number of blocks is denoted *B(R)*. It is also possible to represent the relation size with the number of tuples in *R*. We denote this quantity by *T(R)* (Garcia-Molina et al. 2002). When experimenting with different relation sizes in joins between relations and streams, we use the latter convention i.e. we represent the size of a relation by its number of tuples.

- **Network load**: A network monitoring tool may be used to analyse network traffic on links with varying network load. The network load may range from some kilobits to several megabits, or even gigabits. Consequently, we run all experiments, designed in Section 7.4, with increasing network load. However, a network may be described at different layers e.g. peer-to-peer networks, using the network load to describe the load on each layer. Consequently, the network load differs from layer to layer. When using network load as a factor in our experiment, we consequently refer to network load at the MAC layer i.e. Ethernet headers are included in the network load. We describe network load in terms of Mb/s.

- **Query complexity**: Network-monitoring tasks may be simple or complex. The load queries from e.g. Task 1 and Task 2 are relatively simple, while the TCP connection recognition queries from e.g. Task 4 and Task 7 are relatively complex. A network monitoring tool should be able to process both simple and complex queries. Hence, we investigate how STREAM performs under varying complexities. We divide the queries from our query design section into three categories of different complexity. The category with least complexity includes Task 1, Task 2, Task 8, and Task 11. The next category includes Task 3 and Task 6, while the last category includes Task 4, Task 7, Task 9, and Task 12. It is difficult to decide which of the two latter categories is the most complex in its nature. The queries in the last category create numerous views, and join these views several times. However, the queries in the middle category performs a join between a base stream and a base relation. Besides, the queries in the middle category has the largest window sizes. Thus, depending on what we measure complexity against, the complexity of the two last categories may vary.

- **Number of queries**: One of the requirements in Section 2.3 is that a DSMS should be able to handle several concurrent queries. Thus, we design experiments with only one query executed at the time, and experiments with a varying number of queries executed concurrently.

7.3 Workload

If we capture packets from a network carrying traffic from several protocols, some of this traffic would not belong to TCP/IP. Traffic that is not TCP/IP may contribute to a significant part of the network load. Since Fyaf only sends tuples of TCP/IP headers, collected from TCP/IP packets, to STREAM, the network load that these packets contributes to may be artificially low compared to the actual load on the network. In order to control the factors concerning the input stream e.g. network load, we need to control the network traffic. Consequently, we use a network consisting of only two computers, and generate traffic merely carrying TCP/IP packets. This corresponds to the experiment setup presented in Section 6.3. In Section 6.4.4, we showed that ARP packets contribute to an insignificant part of this traffic. Thus, the number of TCP/IP tuples sent from Fyaf to STREAM is almost equal to the number of packets in the network. In terms of network traffic, such a workload is synthetic. However, with synthetic workloads it is possible to reconstruct the load for each execution of an experiment.

The experiments we design in Section 7.4 include tasks and parameters in a varying manner. However, some parameters are equal for all experiments and are listed below.

- *Number of executions*: We run all experiments five times.
- *Number of connections*: We generate traffic over one TCP connection.
- *Stream duration*: We monitor a stream of TCP/IP header data with duration of 15 minutes. This size does not include the TCP header.
- *Packet size*: We instruct TG to send packets with a TCP segment size of 576 bytes. This does not include TCP and IP headers. This results in a packet size of 628 bytes, including TCP and IP headers, but not Ethernet header, which would add another 14 bytes to the size.

- *STREAM configuration file*: As mentioned in Section 4.4.2, we use one configuration file throughout our analysis. We have changed the memory size and the run time compared to the configuration file included in the STREAM prototype release. The example configuration file used a memory size of 32 MB. We consider this value as far too small when we have a GB of total memory. As we described in Section 6.4.5, other processes than STREAM occupies 2.9 % of total memory. However, we do not allocate all the remaining memory to STREAM, because we are uncertain of how the operating system's kernel utilises memory. In addition, we cannot perform any tests investigating this matter, because of time constraints. Consequently, we introduce a buffer of approximately 20 % and allocate 768 MB, which corresponds to 76.8 %. Thus, STREAM and other user processes together consume approximately 80 % of total memory. Using a run-time value of 10 000 makes STREAM run for approximately ten seconds after receiving the last tuple. A smaller value may lead to termination before all tuples are seen. A larger value results in spending an excess of time on waiting for new tuples. This is not favourable when processing many experiments at different network loads with several iterations. We repeat the values in the configuration file below for convenience.
    - MEMORY_SIZE = 805306368. This is the number of bytes that STREAM allocates. The number corresponds to 768 MB.
    - QUEUE_SIZE = 30. The queue size is given in terms of pages. One page is set to 4096 bytes.
    - SHARED_QUEUE_SIZE = 300. The shared queue size given in terms of pages.
    - INDEX_THRESHOLD = 0.85. This value is similar to the threshold value used in a disk-based linear hash table.
    - RUN_TIME = 10000. The number of times STREAM receives empty tuples from the table source before exiting.

7.4 Experiments

Many of the queries implementing the tasks presented in Section 5.2 are semantically and syntactically similar. In addition, some tasks analyse traffic characteristics e.g. SYN flood attacks, which we cannot easily generate with a traffic-generating tool. Consequently, we do not include all the queries from Section 5.2 in the actual experiments. The queries we include are collected from Task 1, Task 2, Task 3, Task 4, and Task 7. We choose these queries, because they are collected from all the three complexity categories described in 7.2, and we may easily generate traffic to these queries. As mentioned in Section 5.2, all the queries are tested offline. They are all accepted by STREAM with respect to semantics and syntax, and they all produce the expected query answer.

We structure the experiment section as follows: First, we design experiments where STREAM processes one query at the time. Next, we design experiments investigating the performance of STREAM as it processes several queries concurrently. For each experiment within these categories, we design the experiment, present the results, and conclude. We present the results according to the metrics defined in Section 7.1. Concerning relative throughput, we reveal how the relative throughput develops as we increase the network load. In addition, we show how many of the five runs are executed correctly i.e. STREAM does not crash, and consequently contributes to the calculation of relative throughput. For CPU consumption, we examine how CPU consumption develops throughout the 15 minutes stream duration. Calculating the average of CPU consumption from all contributing runs would not give a precise picture of the development, because the consumption may increase or reduce at slightly different points in time. Consequently, we choose one network load and one run to show the percentage of total CPU resources consumed by STREAM. Finally, we reveal how accurate the query answers are by comparing them with the answers that are expected at each network load. As discussed in Section 7.1, it is not straightforward to calculate such an expected answer. The first experiment includes three different tasks. One or more of these tasks are included in all experiments. In addition, all queries are tested offline and they all produce correct query answers. Consequently, we only show accuracy for the three queries involved in the first experiment.

7.4.1 Experiments with Queries Processed Seperately

In this section, we investigate how STREAM performs when processing separate queries. We start by comparing the performance of STREAM when processing queries with different complexity. Next, we investigate the significance of relation sizes in joins between relations and streams. Finally, we investigate how well STREAM optimises queries with respect to pushing projections down the query plan.

Queries with Different Complexity

We start with an experiment that investigates the performance of STREAM when processing three tasks of different complexity.

Design

We choose Task 1, Task 3, and Task 4 from the three complexity categories. Recall that Task 1 measures the load on the network in terms of bytes per second, Task 3 performs a join between a relation and a stream, while Task 4 recognises connections and measures each connection's load. For Task 3, we create a relation consisting of all 65536 port numbers. In the next experiment, we investigate the significance of relation sizes. In this experiment, we use the workload outlined in Section 7.3, and we generate traffic with network loads ranging from 1 Mb/s to 15 Mb/s.

Results

Figure 25(a) presented below reveal the relative throughput for this experiment. We see that the relative throughput is 100% for network loads from 1 Mb/s to 15 Mb/s when processing Task 1. Though not illustrated in this graph, our experiment shows that the relative throughput for Task 1 is close to 100 % for network loads up to 30 Mb/s. In addition, Figure 25(a) indicates that Task 3 and Task 4 have a similar development of relative throughput compared to each other. They have 100 % relative throughput up to 3 Mb/s and 2.5 Mb/s, respectively. Task 4 has 93.32 % relative throughput at 3 Mb/s. However, the relative throughput for Task 3 is 0 % at network loads from 10 Mb/s to 15 Mb/s. The reason for this is that Task 3 does not have any correct runs at these network loads. The number of correct runs, given in percent, are presented in Figure 25(b) below. Task 1 and Task 4 have 100 % correct runs for all network loads from 1 Mb/s to 15 Mb/s. The blue curve, which represents Task 4, covers the red curve, which represents Task 1. The green curve, which represents Task 3, is hidden behind the blue curve from 1 Mb/s to 8.5 Mb/s.

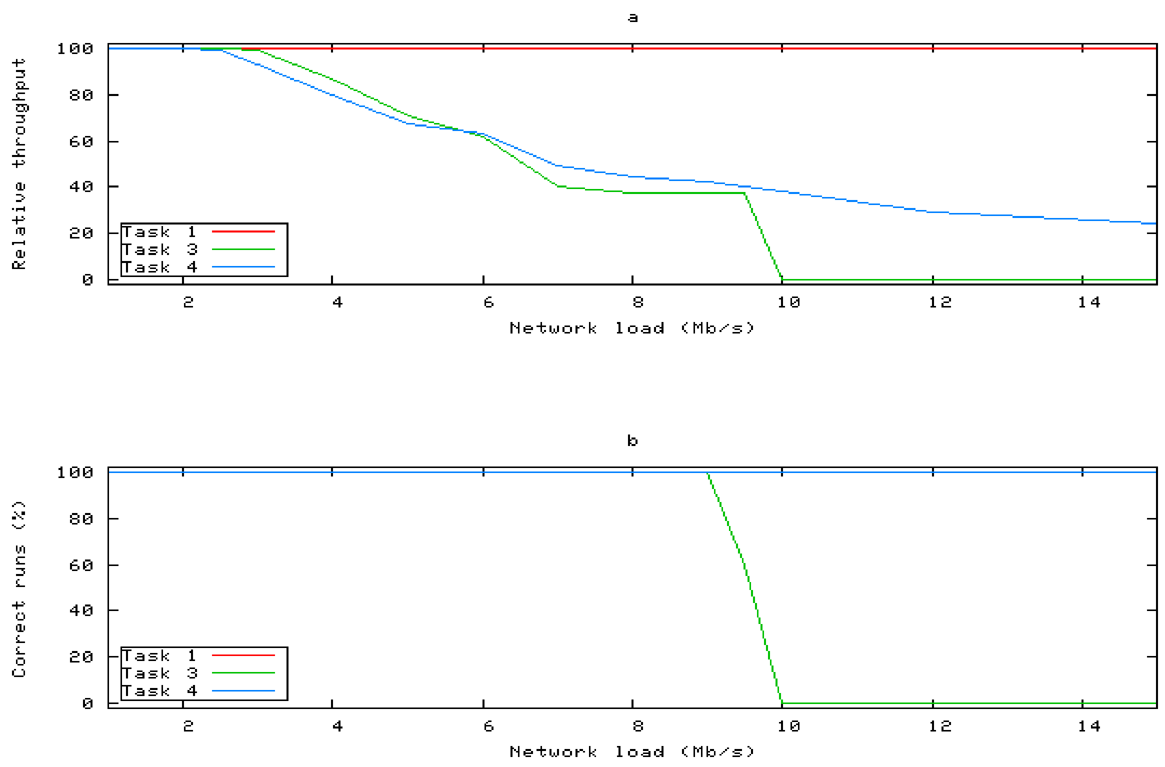


Figure 25. Relative throughput (a) and correct runs (b) for the different queries

Figure 26 presented below reveal the amount of CPU resources STREAM consumes when processing the three tasks. We create the curves from data that Top collects from one of the runs with a network load of 5 Mb/s. We see that STREAM consumes least CPU resources when processing Task 1, which is represented by the curve in Figure 26(a). Figure 26(b) and Figure 26(c) show CPU consumption when processing Task 3 and Task 4, respectively. Figure 26(b) shows that STREAM consumes close to 100 % of the CPU resources, over a period of approximately 350 seconds, when processing Task 3. The curves also show that STREAM consumes CPU resources in a very different manner when processing the three tasks.

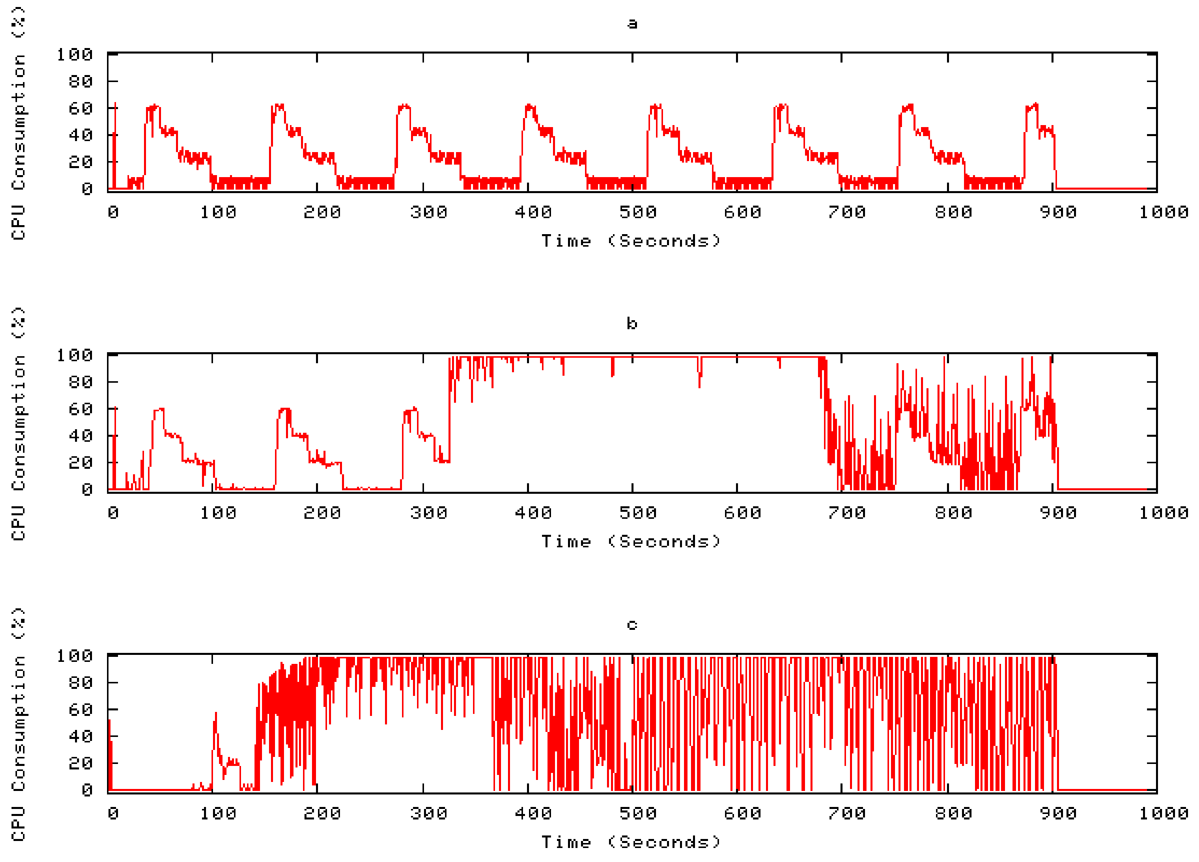


Figure 26. STREAM's CPU consumption when processing Task 1 (a), Task 3 (b), and Task 4 (c) over a network load of 5 Mb/s

Figure 27, presented below reveal the accuracy of the three tasks in this experiment. For each task, we draw two curves. The red curves show the query results produced by STREAM, while the green curves show the expected query results. Task 1 queries the stream to calculate network load measured in bytes per second. We achieve this by summarising total length fields in the IP headers. However, the network load used as factor includes the Ethernet header. Thus, to calculate the expected result at a given network load, we first add 14 bytes, which is the size of the Ethernet header, to the average packet size for different network loads. We use the average packet sizes from Section 6.4.2. Next, we divide the network load by this packet size, which includes the Ethernet header, to obtain the number of packets per second. Following this, we multiply number of packets per second with the average packet size, representing packet sizes at the IP layer, to find the number of bytes per second. However, we add the load constituted by acknowledgements to the last result, because packets transmitted in both directions are captured from the NIC. We calculate the additional load introduced by acknowledgements in Section 6.4.1. We follow a similar procedure to calculate the expected results for Task 3 and Task 4. Indicated by the curves, we comprehend that the difference between the query result and the expected result increase as network load increases, with the query result smaller than the actual result. For Task 1 it is 4.09 % smaller, for Task 3 it is 3.16 % smaller, and for Task 4 it is 2.09 % smaller at 15 Mb/s, 3 Mb/s, and 2.5 Mb/s, respectively.

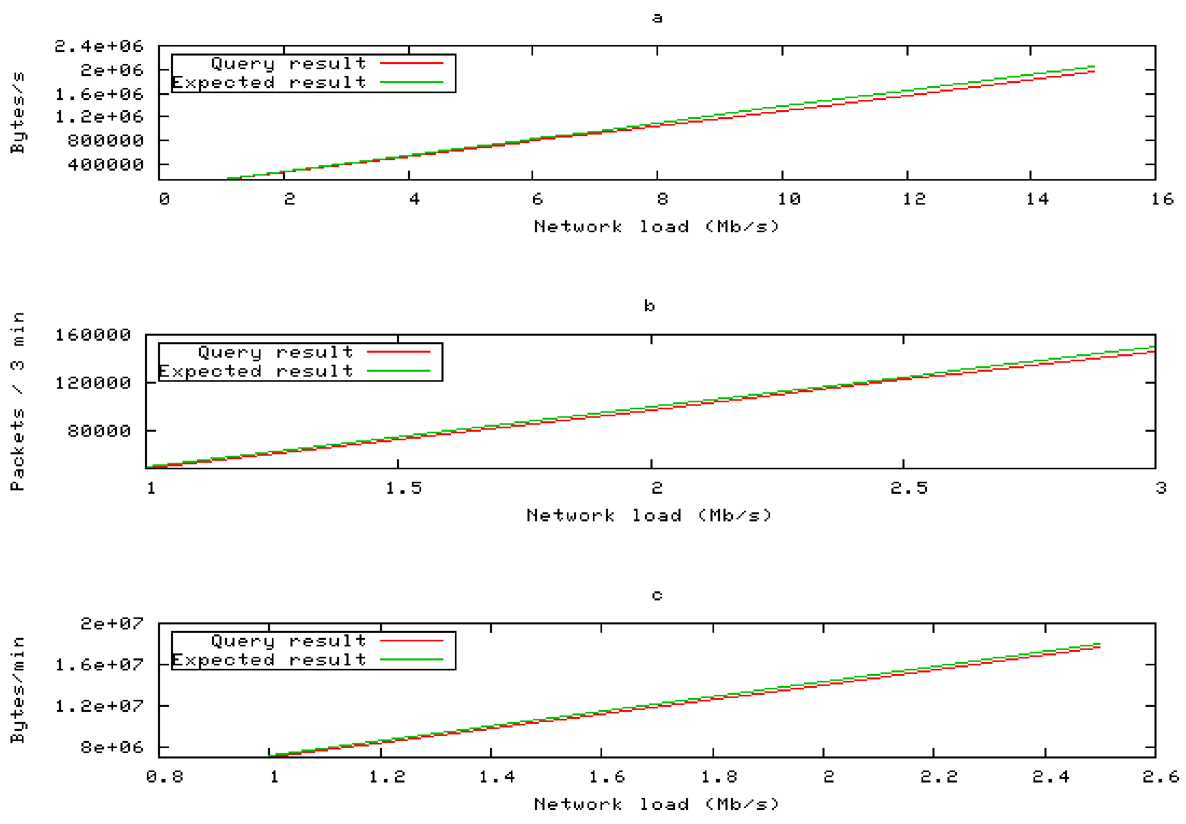


Figure 27. Accuracy of query answers compared to expected results for Task 1 (a), Task 3 (b), and Task 4 (c)

Conclusion

This experiment shows that relative throughput decreases as network load and query complexity increase. In addition, it shows that STREAM crashes when processing Task 3 at 10 Mb/s and above. STREAM neither provides us with information about its problems through its log file, nor do we find anything in the STREAM source code revealing any problems. Consequently, we do not know what internal mechanisms cause it to crash. However, we assume that the problems are due to heavy load, because it crashes at 10 Mb/s and above. We only show CPU consumption at 5 Mb/s. At this network load, we see that STREAM's CPU consumption increases as the complexity of queries increases. When designing this experiment, we describe the uncertainties with calculating a query answer to expect e.g. the usage of average packet sizes. In addition, we calculate accuracy for network loads with relative throughput down to 98 %. By taking these factors and the results shown in Figure 27 into consideration, we conclude that STREAM produces accurate query answers in this experiment.

Joining Streams with Relations of Different Sizes

In this experiment, we investigate the impact of different relation sizes when STREAM processes the query from Task 3, which includes a join between a relation and a stream. Recall that the join in this query is a theta-join, where the condition is equality of port numbers in the relation, and destination port numbers in the stream. All tasks in this experiment are executed from 1 Mb/s to 10 Mb/s.

Design

Since port numbers are defined as data type `short`, which consists of 16 bits, there are 65536 different port numbers. We create three relations of different sizes. The first relation consists of all 65536 port numbers, ranging from zero to 65535. The second relation consists of 1024 port numbers, ranging from 60000 to 61023. The third relation consists of only a single port number namely 60010. We refer to these three different versions of Task 3 as Task 3, Task 3.1, and Task 3.2, respectively.

A careful reader may observe that it is unnecessary with a relation containing all 65536 port numbers. If all ports are of interest, a more effective solution is to query the stream with a `GROUP BY` clause and the necessary aggregation. However, optimisation is not an issue in this experiment. Besides, the semantics of the queries should be equal for the comparison to make any sense.

The stream contains two different port numbers, because acknowledgments are part of TCP. Packets that are sent from computer A to computer B carries 60010 as their destination port number. In the Linux implementation on these computers, the client side of a TCP connection allocates port numbers from 1024 to 29999. Consequently, packets transferred in the other direction, from B to A, have destination port number in this range. Since Task 3 contains all port numbers, the join matches two destination port numbers in the stream, one in each direction. Task 3.1 and Task 3.2, on the other hand, only match one of the two destination port numbers in the stream.

Results

Figure 28(a) presented below shows the relative throughput when processing the different tasks included in this experiment. It reveals that the relative throughput is almost equal for the three different relation size implementations of the query in Section 5.2.3. Actually, the curves are so similar that only one curve is visual throughout much of the graph. Task 3.1 and Task 3.2 have relative throughput of approximately 100 % up to 2.5 Mb/s, while Task 3 has relative throughput of approximately 100 % up to 3 Mb/s. For all tasks, the relative throughput drops from approximately 37 % at 9.5 Mb/s to 0 % at 10 Mb/s. The reason for this is that the three tasks do not have any correct runs at 10 Mb/s, as shown in Figure 28(b) below. In this graph, the blue curve covers the green curve from 1 Mb/s to 9 Mb/s, while the red curve is hidden from view behind the blue curve up to 8 Mb/s, and behind the green curve from 9 Mb/s to 10 Mb/s.

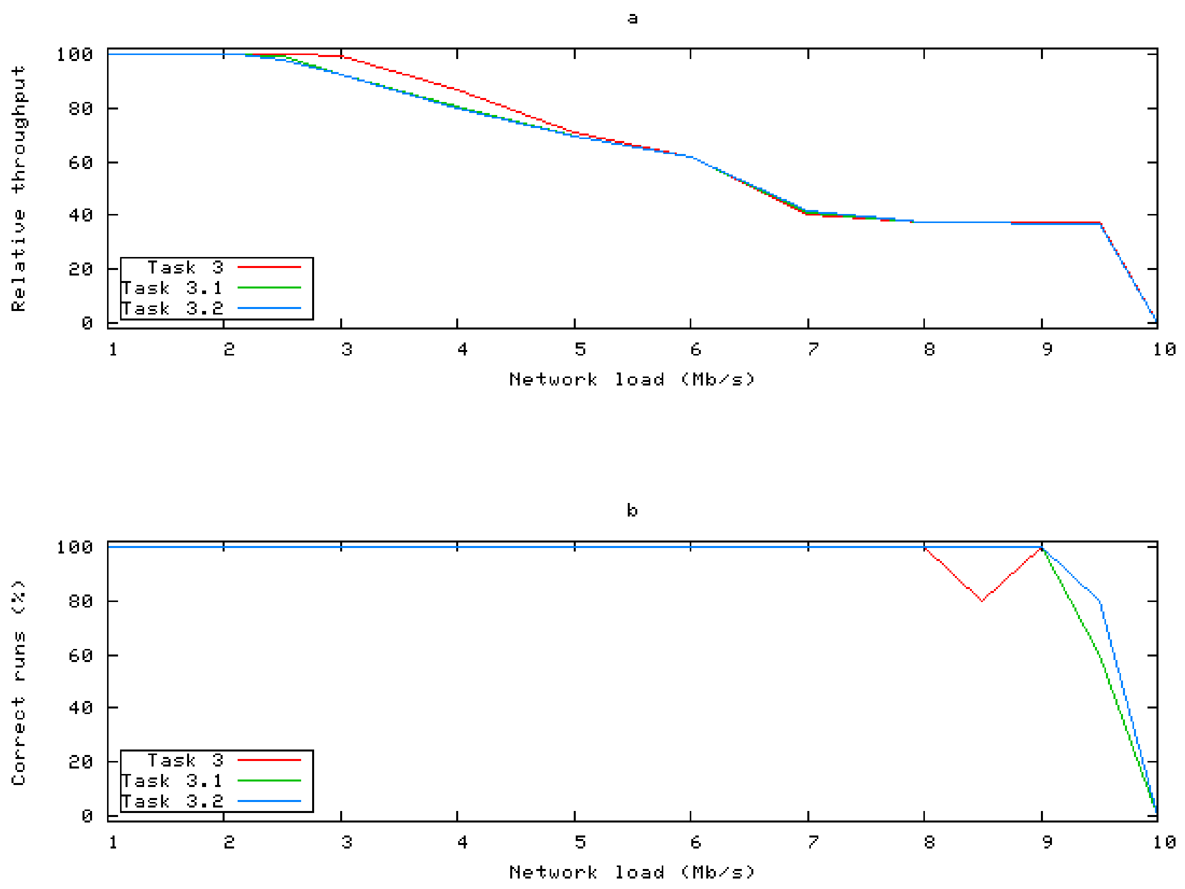
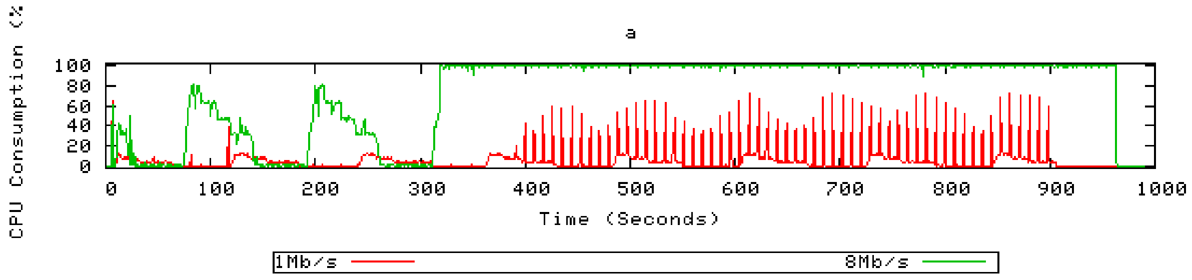
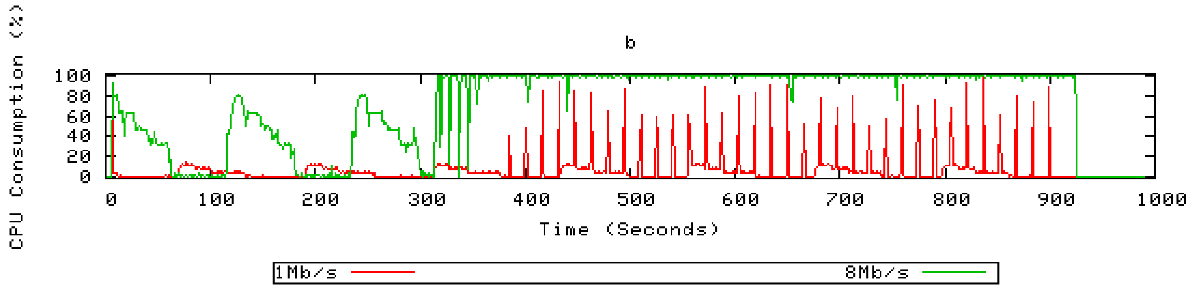


Figure 28. Relative throughput (a) and correct runs (b) for the different queries

Figure 29 below, reveals STREAM's CPU consumption while processing the data streams according to the three queries in this experiment. In each of the three figures, we draw curves representing CPU consumption at 1 Mb/s and 8 Mb/s, using a red and a green curve, respectively. Figure 29(b), which represent Task 3.1, and Figure 29(c), which represents Task 3.2, are rather similar. Figure 29(a), which represents Task 3, differs from the others. STREAM consumes most CPU resources when processing Task 3, which use the relation containing most tuples. With a network load of 1 Mb/s the CPU consumption has a sudden increase after approximately 400 seconds for all three tasks. With a network load of 8 Mb/s the CPU consumption increases from 0 % to 100 % after approximately 300 seconds.





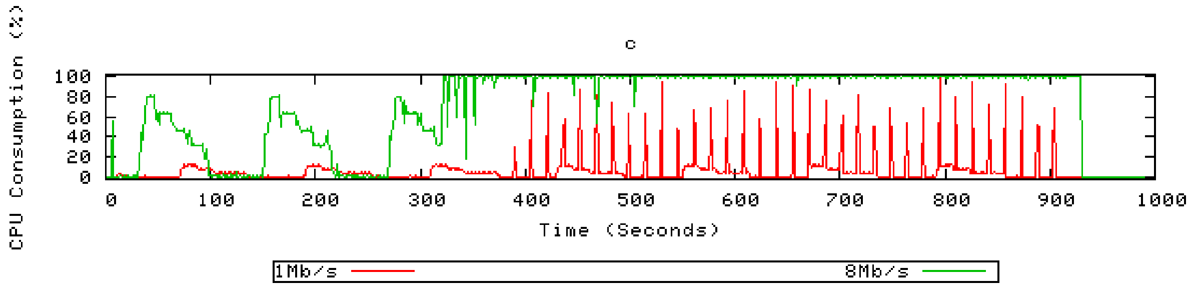


Figure 29. STREAM's CPU consumption when processing Task 3 (a), Task 3.1 (b), and Task 3.2 (c) over network loads of 1 Mb/s and 8 Mb/s

Conclusion

With respect to relative throughput, this experiment demonstrates that the size of the relation does not influence the performance of STREAM when processing the query from Task 3. In addition, it illustrates that relative throughput decreases as network load increases. With respect to CPU consumption, this experiment shows that STREAM consumes more CPU resources as the relation size increase and that CPU consumption increase as we increase the network load. The reason why CPU consumption suddenly increases after 300 seconds at 8 Mb/s may be that these queries perform a join with a five minutes window over the stream. We have no explanation to why there is an increase in CPU consumption after 400 seconds at 1 Mb/s.

Optimising Queries by Pushing Projections Down the Query plan

In the following experiment, we investigate the impact of pushing projections down the query plan. In order to accomplish this, we compare the execution of two different script files. In one of the script files, we use the query as designed in Section 5.2.4 and leave all optimisation to STREAM. In the other script file, we try to optimise by rewriting the query in the script file. In the next section, we investigate other optimisation techniques. We use common descriptions of these script files in all experiments investigating optimisation to avoid confusion. When we use the queries without any optimising efforts, we refer to the scripts as "System optimisation". We refer to the script we try to optimise as "Optimise by hand".

Design

As mentioned in Section 4.3.5, the DSMS may find itself in a state where it is either CPU or memory limited. To avoid the occurrence of such states, it is important to maintain a best possible utilisation of these resources. The query compiler transforms a query several times, as noted in Section 3.2. When creating a physical query plan out of the logical query plan, the query compiler makes use of several algebraic expressions for improving the query plan (Garcia-Molina et al. 2002). One group of such algebraic expressions is related to pushing projections as far down the query plan as possible. By pushing projections down the query plan, we may reduce the tuple sizes, which in turn may lead to less memory consumption. To illustrate the impact of pushing projections we consider the following query.

```
SELECT sourceIP, destIP, COUNT(*)
FROM S [RANGE 10 MINUTES]
GROUP BY sourceIP, destIP
```

A tuple created by Fyaf, consisting of TCP and IP header values from a packet sent form A to B, typically looks like:

```
4, 5, 0, 628, 18681, 2, 0, 64, 6, 25901,
129.240.67.60, 129.240.67.65, 0, 5862, 60010,
993126829, 1870095210, 8, 0, 0, 1, 1, 0, 0, 0, 1460,
28707, 0, 0101080a2a3bae1391b5b6ec
```

The attribute values are represented in ASCII when entering STREAM, and excluding commas and spaces, this tuple occupies 119 bytes. In STREAM, the attributes are transformed to their respective data types according to how they are defined. The tuple consists of 19 `integers`, two `char(81)`, two `char(16)`, and six `char(2)`. This results in 282 bytes when representing the tuple in STREAM. Consider packets with a packet size of 1500 bytes, sent at 10 Mb/s. Then 873,8113 packets are sent each second. When transformed to tuples of 282 bytes, these packets constitute 0.235 MB each second. Over a ten-minute window, this results in 141 MB. To optimise the query above, we first create a view that projects all necessary attributes. This view is then queried to obtain the semantics of the example.

```
PROJECT (sourceIP, destIP):
SELECT sourceIP, destIP
FROM S


SELECT sourceIP, destIP, COUNT(*)
FROM PROJECT [RANGE 10 MINUTES]
GROUP BY sourceIP, destIP
```

With such an approach, the tuples in the final query occupies 32 bytes. With the parameters given above, this results in 16 MB over the ten-minute window. In this example, we save 125 MB by pushing the projection down the query plan.

In this experiment, we choose to optimise Task 4 by pushing projections down the query plan. We create a view in a similar manner as above. All queries in Task 4 then query this view instead of S. The attributes projected in the view are `ip_headerLength`, `totalLength`, `sourceIP`, `destIP`, `sourcePort`, `destPort`, `seqNum`, `ackNum`, `tcp_headerLength`, `ACK`, `SYN`.

We expect that the query compiler in STREAM optimises the query in a best possible manner. By performing this experiment, we try to clarify the impact of how a user writes a query. If STREAM optimises the system by pushing projections down the query plan, the introduction of a new view may only introduce more overhead to the system, and consequently reduce performance. When performing the experiment, we use the workload described in Section 7.3, and we generate traffic with network loads ranging from 1 Mb/s to 15 Mb/s.

Results

Figure 30(a) below reveals the development of relative throughput when processing the two queries with increasing network load. The two queries have almost equal relative throughput on all network loads from 1 Mb/s to 15 Mb/s. For both queries, the relative throughput is close to 100 % up to 2.5 Mb/s. They have a relative throughput of approximately 50 % at 7 Mb/s. Figure 30(b) below, indicates that STREAM execute without any crash from 1 Mb/s to 15 Mb/s. The green curve covers the red curve in this graph.

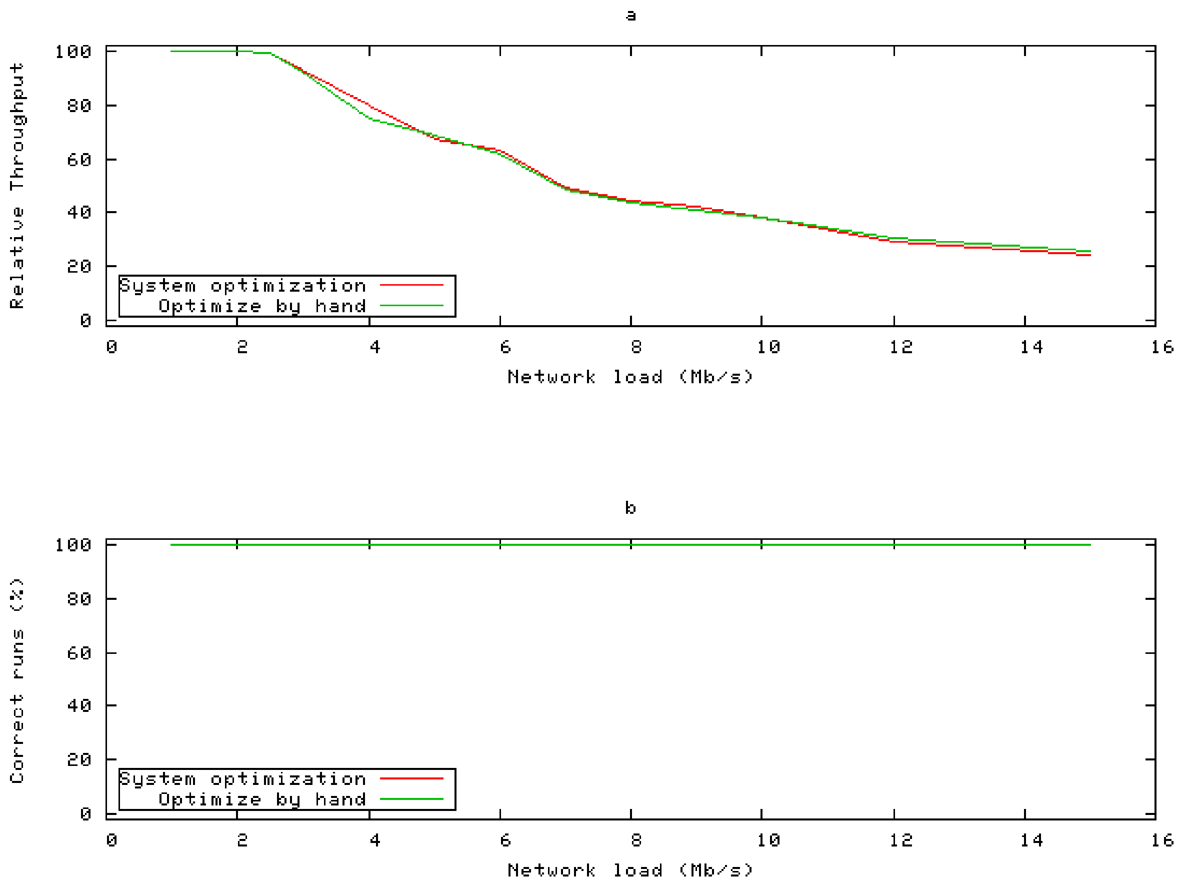


Figure 30. Relative throughput (a) and correct runs (b) for the different queries

Figure 31 below shows STREAM's CPU consumption when processing the two different queries. The curves are created from one of the runs at 5 Mb/s. Figure 31(a) shows CPU consumption when processing the original task that do not have any pushed projections. Figure 31(b) reveals CPU consumption when processing the new version of Task 4 that have projections pushed down the query plan. The two curves are almost exactly equal. There is an increase in CPU consumption after approximately 140 seconds, and it stays relatively high the next 230 seconds. After 370 seconds, the CPU consumption shifts from 0 % to 100 % for the rest of the executions.

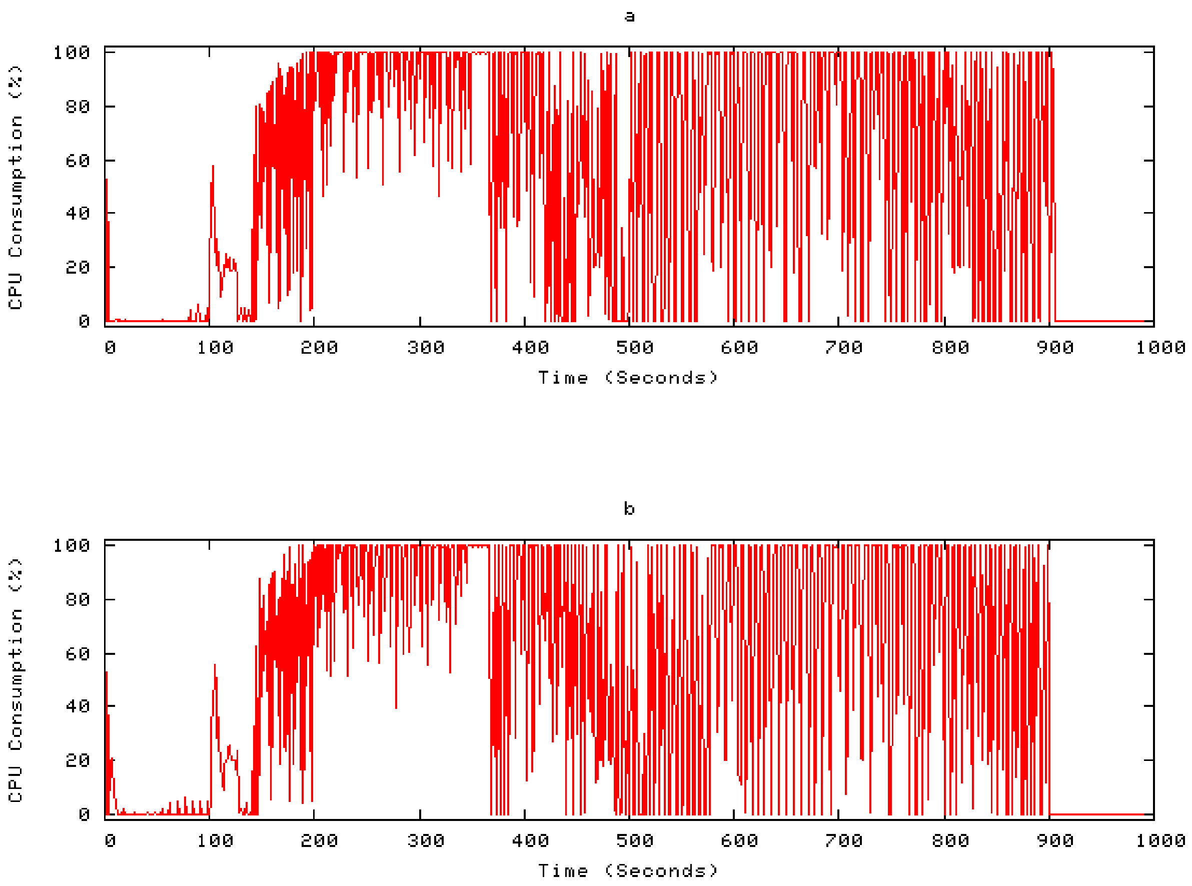


Figure 31. STREAM's CPU consumption when processing query optimised by system (a) and query optimised by hand (b) over a network load of 5 Mb/s

Conclusion

Optimising the query from Task 4 by pushing projections into a view has no effect on STREAM's performance with respect to relative throughput and CPU consumption. It would be interesting to investigate how STREAM consumes memory when processing the two queries, because pushing projections down the query plan primarily reduces memory consumption. This is not possible, because we treat STREAM as a black box. Consequently, we cannot add such monitoring functionality. Moreover, we cannot use the monitoring programs for this purpose, because STREAM allocates memory statically.

7.4.2 Experiments with Queries Processed Concurrently

In this section, we design different experiments investigating system behaviour as we introduce concurrent execution of queries. First, we design two experiments investigating how well STREAM shares resources between queries with common sub-expressions. Recall from Section 4.3.3 that STREAM shares queues and synopsis when queries with common sub-expression appear. Next, we investigate STREAM's performance as we increase the number of concurrent queries.

Sharing Resources

When continuous queries contain common sub-expressions, it is possible to optimise execution by sharing resources and computation within their query plans. Two such resource-sharing techniques are queue sharing and synopsis sharing. In this section, we perform two experiments that investigate how well these techniques work within STREAM. To accomplish this experiment we create two different script files. We compare the executions of these script files against each other. In the first script file, we include queries that have semantically equal sub-expressions. However, we give these sub-expressions different names to distinguish them from one another and let STREAM optimise. In such a scenario, STREAM should share both queues and synopsis based on the common semantics of these queries' sub-expressions. In the second script file, we optimise the queries by hand through assembling the common sub-expressions into one sub-expression used by all queries. When performing these experiments, we use the common workload, as described in Section 7.3, and we generate traffic with network loads ranging from 1 Mb/s to 15 Mb/s.

Design

Sharing Resources between the Different Versions of Task 3

In this experiment, we implement script files that include the three different versions of Task 3. As explained in Section 5.2.3, we create a view to solve this task, because the total number of attributes from the two sources in this join is too high. Recall that the three versions of Task 3 differ from each other based on the relation size.

Sharing Resources between Task 4 and Task 7

In this experiment, we implement script files where Task 4 and Task 7, which both recognise connections, are processed concurrently. Recall that Task 4 finds the number of bytes that have been exchanged on each connection during the last minute, while Task 7 finds the network load measured in connections per minute. The sub-expression these tasks have in common includes the three views, Syn, Synack, and Conn, which are involved in recognising established connections.

Results

Sharing Resources between the Different Versions of Task 3

Figure 32 presented below shows the relative throughput and the number of correct runs for this experiment. Figure 32(a) indicates that the relative throughput is equal when processing the two different script files. They have 100 % relative throughput up to 1.5 Mb/s. The relative throughput starts to level off at 50 %. This happens at approximately 4 Mb/s. At 15 Mb/s the relative throughput is approximately 35 %. In Figure 32(b), showing the number of correct runs, reveals that all runs contribute to the relative throughput up to a network load of 12 Mb/s. From 12 Mb/s to 15 Mb/s the number of correct runs decreases to 40 % for the execution that is optimised by the system. The green curve covers the red curve from 1 Mb/s to 12 Mb/s.

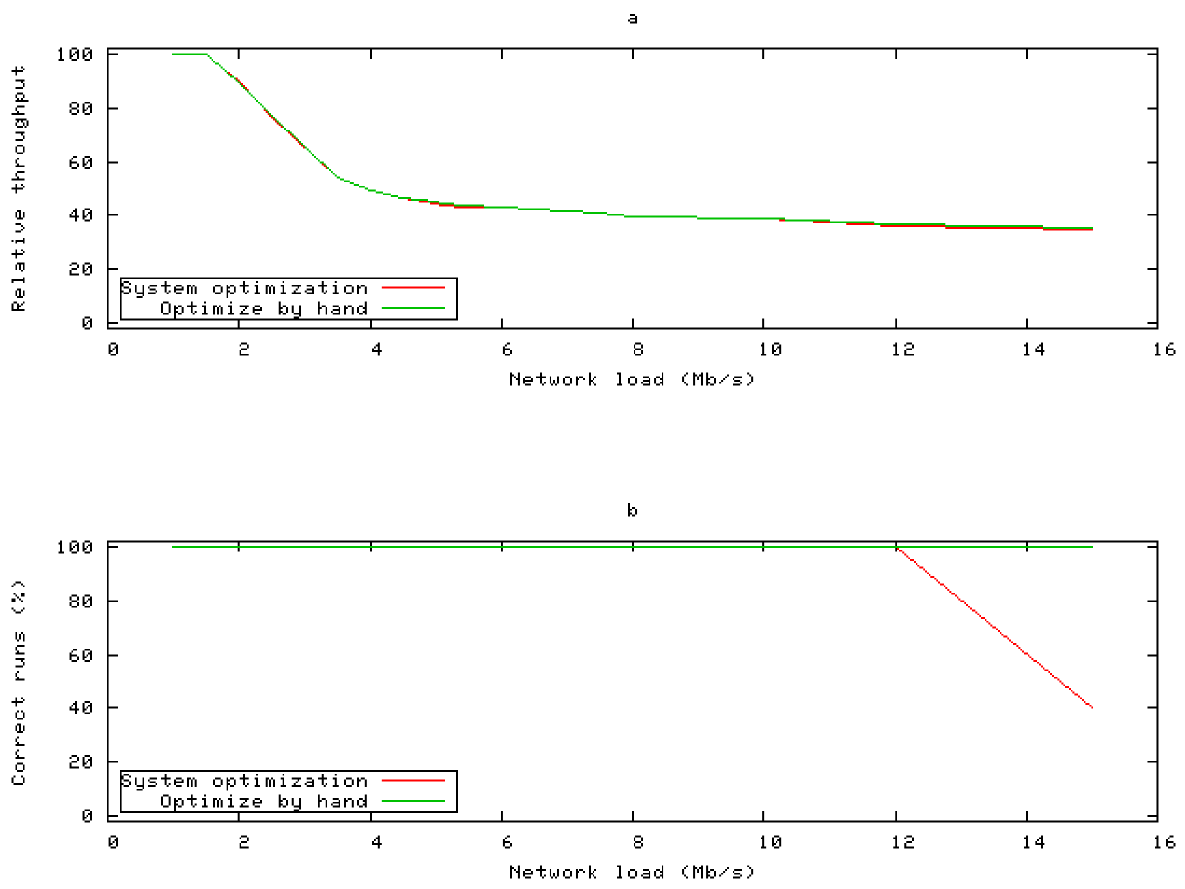
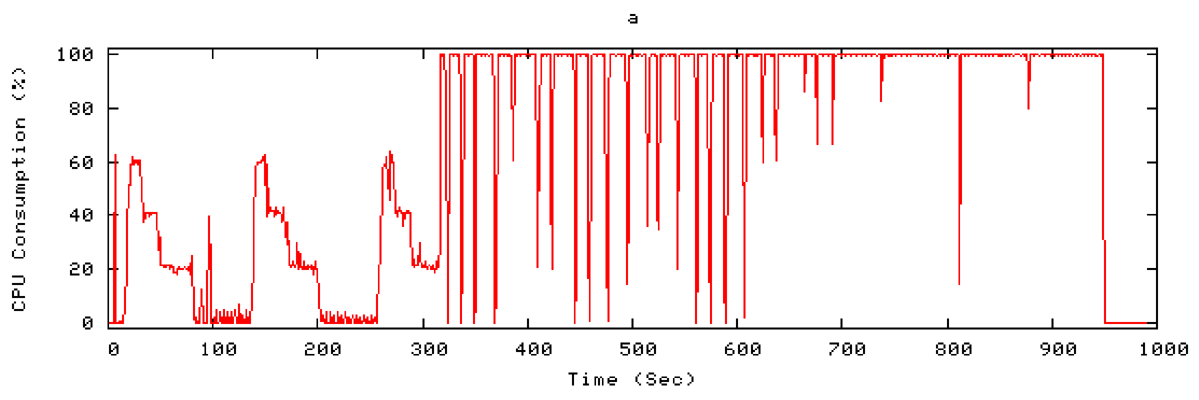
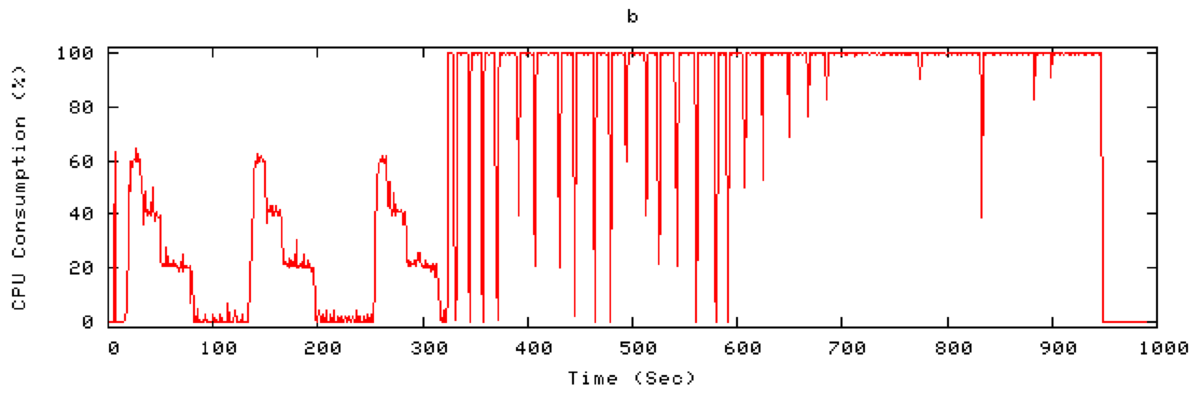


Figure 32. Relative throughput (a) and correct runs (b) for the two versions of the queries

Figur 33 reveals the amount of CPU resources STREAM consumes when processing the two different script files. Both figures are created from one of the runs at 5 Mb/s. Figur 33(a) represents CPU consumption when processing the script where optimisation was left to STREAM, while Figur 33(b) represents the script that we optimise by hand. The two curves are almost equal. They have a steep rice in CPU consumption after approximately 300 seconds. From 300 seconds to 600 seconds they have 100 % CPU consumption, with sudden drops that last a couple of seconds approximately every 15 second. After approximately 600 seconds, the CPU consumption stays more constant at 100 %. The CPU consumption drops to 0 % at 950 seconds, indicating that the last element in the input stream has arrived within STREAM.





Figur 33. STREAM's CPU consumption when processing queries optimised by system (a) and queries optimised by hand (b) over a network load of 5 Mb/s

Sharing Resources between Task 4 and Task 7

Figure 34 below presents the relative throughput and the number of correct runs in this experiment. The red curve represents executions where STREAM optimises the queries. The green curve represents executions where we optimise the queries by hand. From Figure 34(a), which shows relative throughput, we see the relative throughput is higher when we optimise by hand. With system optimisation, the relative throughput is approximately 100 % up to 2 Mb/s, while the relative throughput is approximately 100 % up to 2.5 Mb/s when we optimise by hand. At 2.5 Mb/s the system optimisation has a relative throughput of 89.75 %. We find the largest difference between the two curves at a network load of 5 Mb/s, where system optimisation and the optimisation by hand have relative throughputs of 53.25 % and 66.26 %, respectively. The curves have almost equal values from 10 Mb/s, they have a relative throughput of approximately 30 %. At 15 Mb/s, they have a relative throughput of approximately 20 %. Figure 34(b) shows the number of correct runs. The green curve covers the red curve from 1 Mb/s to 12 Mb/s. For system optimisation the number of correct runs drops to 80 % at 15 Mb/s, while all runs are correct when optimising by hand.

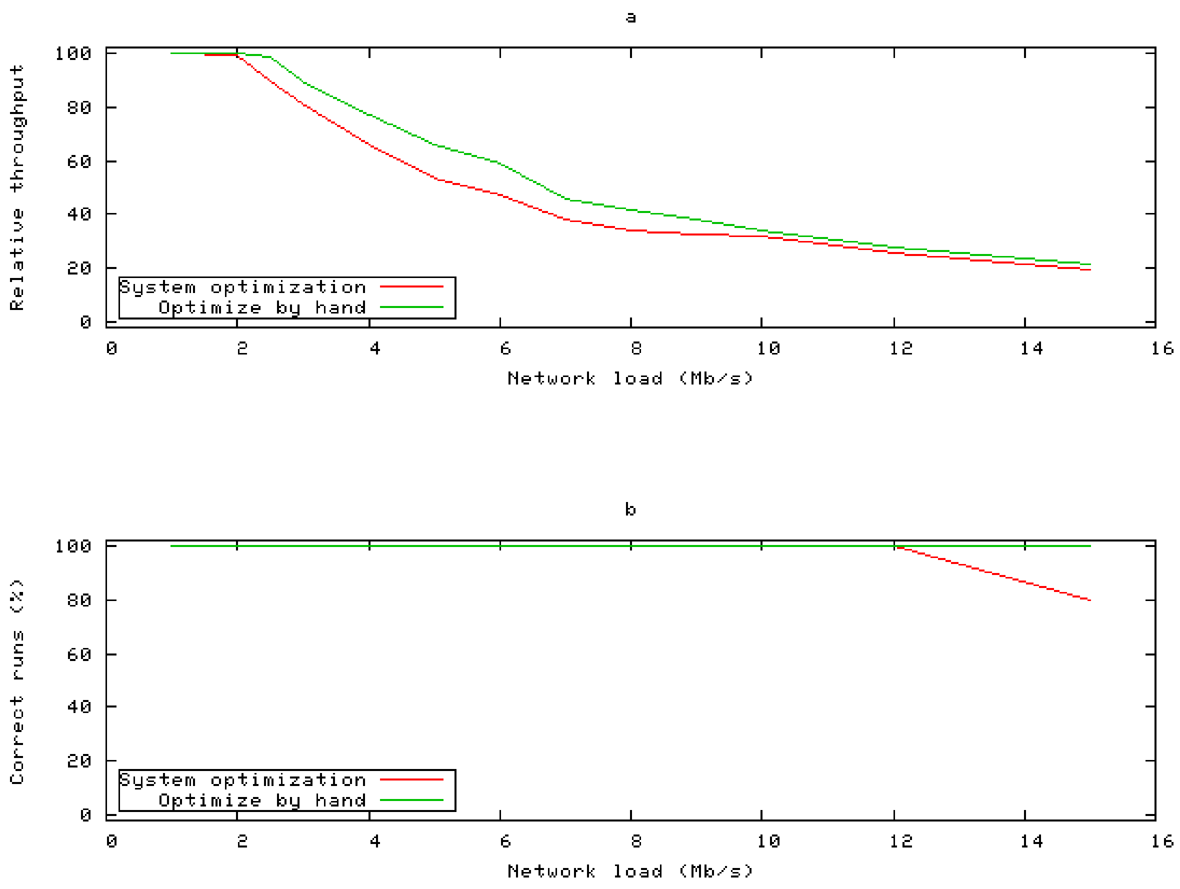
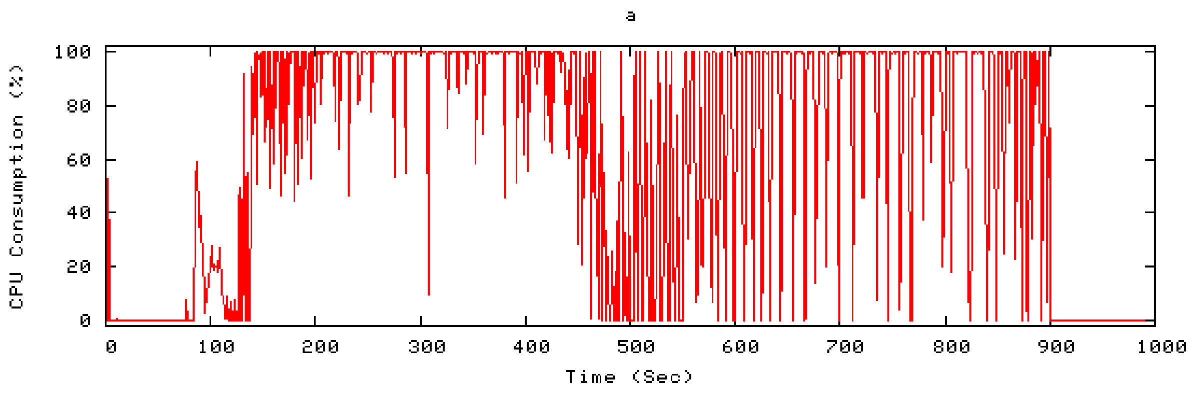


Figure 34. Relative throughput (a) and correct runs (b) for the two versions of the queries

Figure 35 below illustrates the CPU consumption when processing the queries at a network load of 5 Mb/s. Figure 35(a) and Figure 35(b) show CPU consumption when STREAM optimises and when we optimise by hand, respectively. Both curves show a steep rice in CPU consumption after approximately 130 seconds. System optimisation and optimisation consumes close to 100 % of the CPU resources up to 450 seconds and 360 seconds, respectively. Throughout the rest of the runs their CPU consumptions shifts from 100 % to lower percentages, with system optimisation having longer periods with a CPU consumption of 100 %.



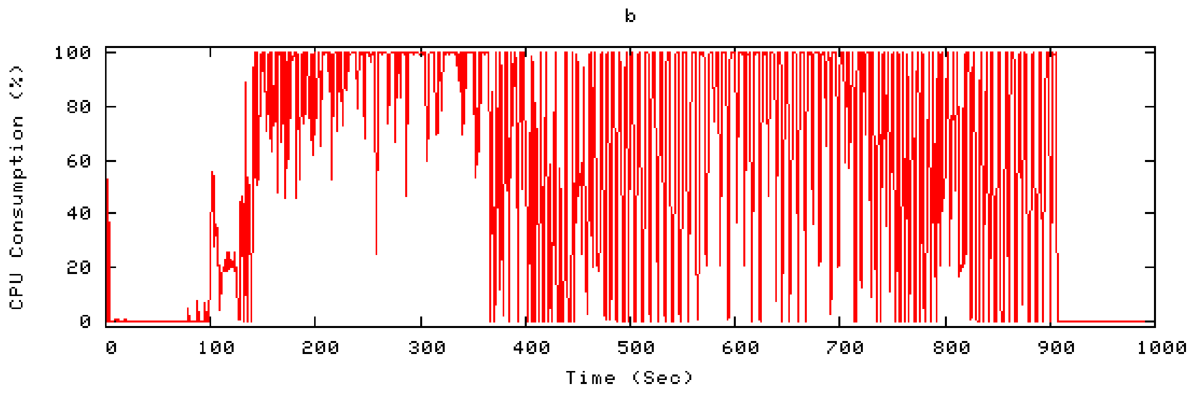


Figure 35. STREAM's CPU consumption when processing queries optimised by system (a) and queries optimised by hand (b) over a network load of 5 Mb/s

Conclusion

In the first experiment, there is no difference in STREAM's performance when processing queries optimised by system and queries optimised by hand. In the second experiment, we increase the number and complexity of the shared resources. This results in a difference in performance, where STREAM has a better performance when optimising by hand then when it optimises itself. Section 4.3.3 described that STREAM shares sub-expressions by introducing stores and stubs. In order to share the resources in the second experiment, STREAM makes use of one store for the three different types of views and one stub for each view. Thus, in order to share these sub-expression, STREAM requires three stores and nine stubs. This results in more overhead and resource consumption by STREAM, which in turn leads to a drop in performance. Consequently, a user may improve STREAM's performance by optimising the script files, at least when several complex queries, with several complex sub-expressions in common, are executed concurrently.

Concurrency with Varying Number of Queries and Complexity

Design

In this experiment, we investigate the performance of STREAM by processing several different script files with a varying collection of queries. We vary the script files by including queries with different complexity. In addition, we vary the number of concurrent queries in these files. Two of these script files include combinations of queries from the two previous experiments. We use the script files where queries use one common sub-expression. We refer to the different scripts by the word "Script" and a number to avoid listing all tasks when referring to them. The different combinations of tasks are as follows:

1. **Script 1**: This script includes Task 3, Task 3.1, and Task 3.2. This combination is adapted from Experiment 4.
2. **Script 2**: This script includes Task 4 and Task 7, and is adapted from Experiment 5.
3. **Script 3**: This script in includes Task 1, Task 2, Task 3, Task 3.1, and Task 3.2.
4. **Script 4**: This script includes Task 1, Task 2, Task 3, Task 3.1, Task 3.2, Task 4, and Task 7.

In addition, we are interested in combinations including Task 9 e.g. the combination of Task 4, Task 7, and Task 9. We try this combination of queries with both separate and common sub-expressions. However, when executing these scripts STREAM exits with the message “Error: Unknown error: -1.” By investigating the source code, we find that this is due to the constants `MAX_OUT_BRANCHING` and `MAX_STUBS`, respectively. Task 9 alone does not cause these errors, but they are rather caused by the combination of many complex queries. We do not try any new combinations with Task 9, because of these errors and because we already use two queries from its complexity category.

When performing this experiment, we use the common workload from Section 7.3, and we generate traffic with network loads ranging from 1 Mb/s to 15 Mb/s.

Results

Figure 36 below indicates the relative throughput and number of correct runs for this experiment. Figure 36(a) shows that the relative throughput for Script 2, which includes Task 4 and Task 7, is close to 100 % for network loads up to 2.5 Mb/s. The network load is 100 % only at 1 Mb/s when processing Script 4, which includes all seven tasks. Script 1, Script 3, and Script 4 has a relative throughput of approximately 50 % at 4 Mb/s, where the curves starts to layer off. The curve representing Script 2, layers off at 6.5 Mb/s where the relative throughput is approximately 50 %. Relative throughput for Script 3 drops from 34.55 % at 12 Mb/s to 0 % at 15 Mb/s. At 15 Mb/s Script 1, Script 2, Script 3, and Script 4 have relative throughputs of 35.36 %, 21.70 %, 0 %, and 12.80 %, respectively. The number of correct runs is presented in Figure 36(b). At the network loads where only the purple curve is visible, the other curves are hidden in view behind it. The number of correct runs for Script 3 drops from 100 % at 12 Mb/s to 0 % at 15 Mb/s, while the number of correct runs for the other scripts is 100 % for all network loads.

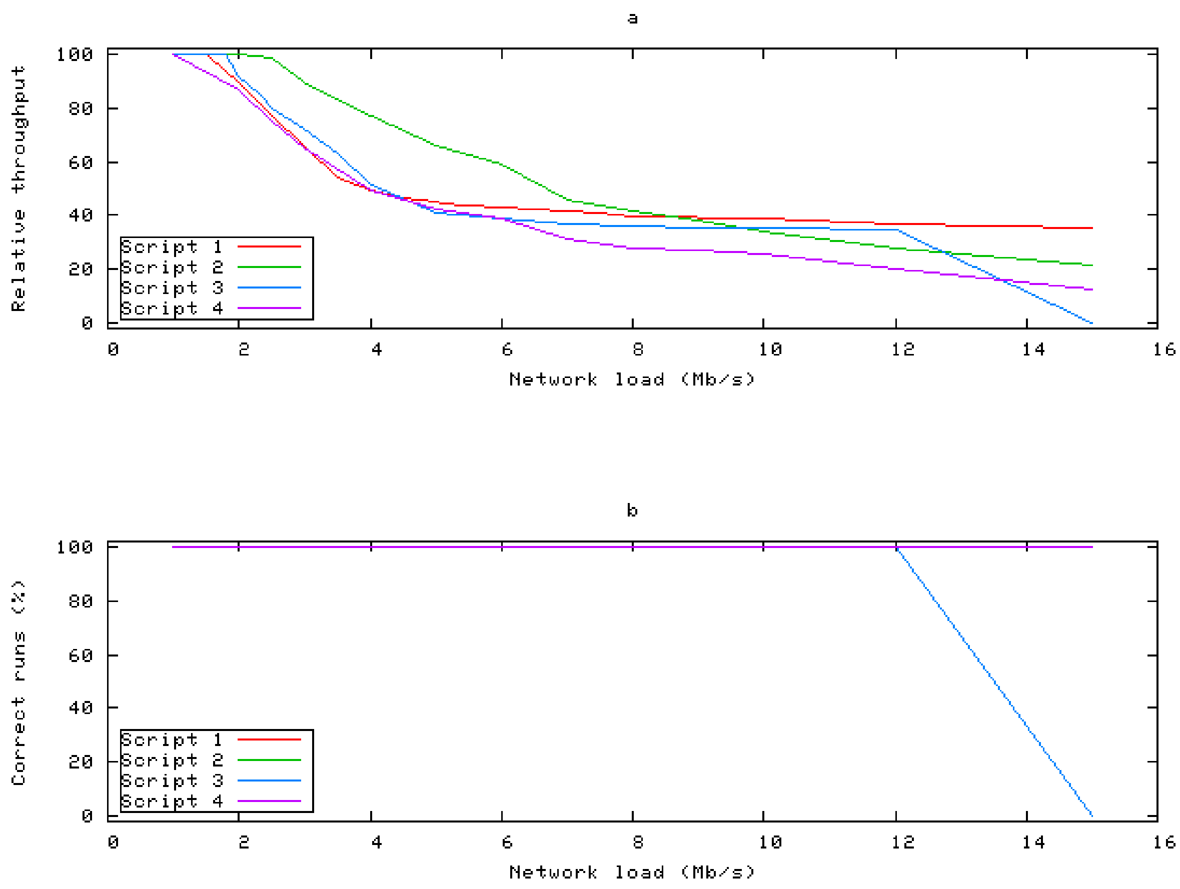


Figure 36. Relative throughput (a) and correct runs (b) when processing the different scripts

Figure 37 below reveals STREAM's CPU consumption in this experiment. Figure 37(a), which represents Script 1, and Figure 37(c), which represents Script 3, both show an increase in CPU consumption after approximately 300 seconds. After that, CPU consumption is close to 100 %. Script 3 leads to more CPU consumption than Script 1. Figure 37(b), which represents Script 2, and in Figure 37(d), which represents Script 4, indicate an increase in CPU consumption after approximately 120 seconds. Script 4 leads to more CPU consumption that Script 2. Script 4 consumes more resources at the beginning of the execution. Script 3 consumes most after 300 seconds. The same difference applies to Script 1 and Script 2. Script 2 consumes most CPU resources in the beginning, while Script 1 consumes most CPU resources from 300 seconds. We cannot decide which script file consumes most CPU resources and which script file consumes the least CPU resources, because the CPU consumption is different over time. However, Script 3 and Script 4 consume more CPU resources than Script 1 and Script 2.

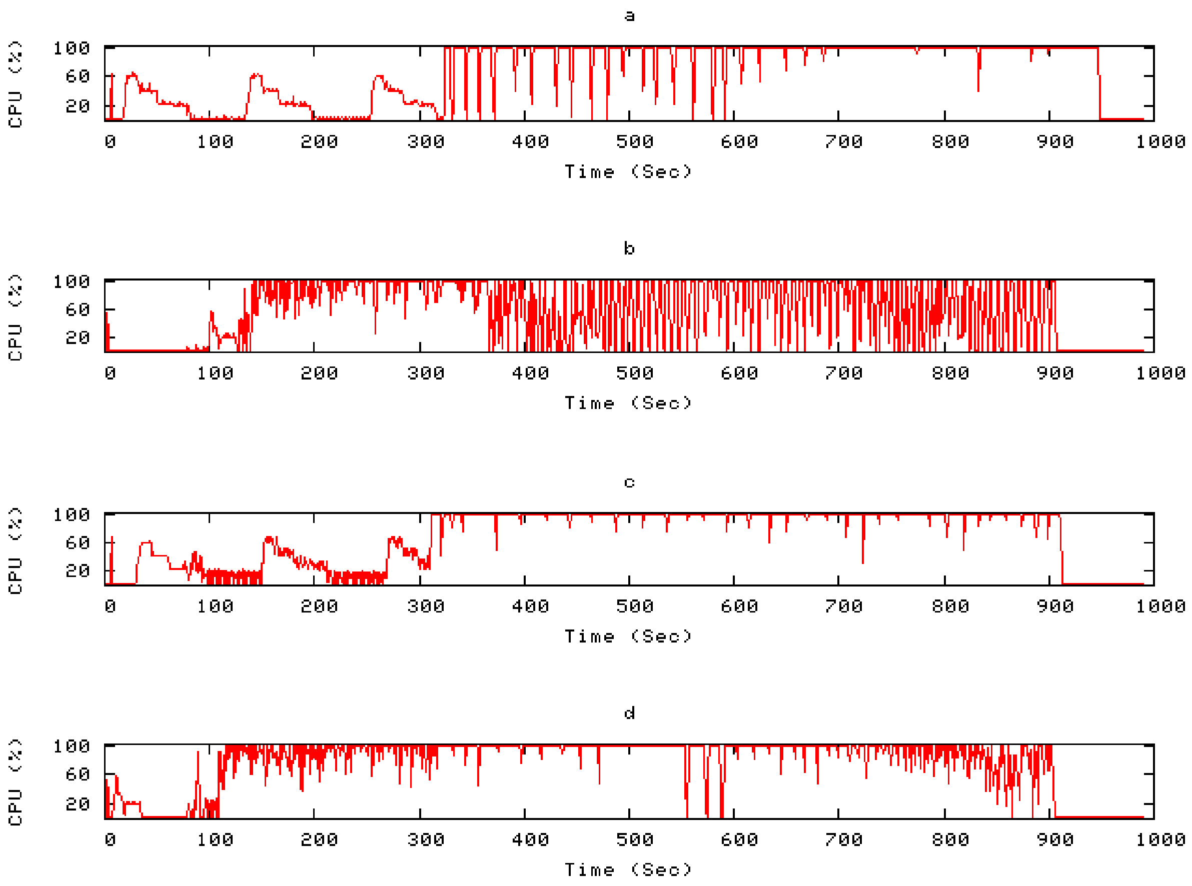


Figure 37. STREAM's CPU consumption when processing Script 1 (a), Script 2 (b), Script 3 (c), and Script 4 (d) over a network load of 5 Mb/s

Conclusion

With respect to both CPU consumption and relative throughput, we see that STREAM's performance drops as the number of concurrent queries increase. This is also the case when we compare these results to experiments where tasks are executed separately. However, all the versions of Task 3 have a relative throughput of 0 % when executed separately. Thus, STREAM performs better up to 10 Mb/s when processing Task 3 separately, with respect to relative throughput. From 10 Mb/s it performs better when Task 3 is executed concurrently with other tasks. One surprising result in this experiment is that the relative throughput for Script 3 drops to 0 % at 15

Mb/s, because there are no correct runs. This is not surprising in itself, but the relative throughput of Script 4, which includes two queries in addition to all the queries from Script 3, does not drop to 0 % at 15 Mb/s.

7.5 Discussion and comparison

When processing tasks separately, STREAM has a relative throughput of 100 % up to approximately 3 Mb/s. The only exception is when executing Task 1, where STREAM has a relative throughput of 100 % up to approximately 30 Mb/s. With respect to processing, this shows that there are large differences in complexity between the three tasks processed separately. Another complexity factor is window size. Task 1 uses a view to summarize total lengths over a one-second window and then query this view to calculate average over a ten-second window. Consequently, the view produces one tuple every second. This results in ten tuples in the final query's window. These numbers are independent of the network load. Task 3 uses a window size of five minutes to unblock the input stream. The number of tuples in this window increases as the network load increases. Given a network load of 1 Mb/s and a packet size of 1500, the five-minute window contains approximately 26 214 tuples after the first five minutes have passed. Thus, window size is an important factor with respect to query complexity.

When processing two or more queries concurrently, STREAM has a relative throughput of 100 % from one to approximately 2 Mb/s or 3 Mb/s. However, this depends on the number of concurrent queries, and the complexity of the concurrent queries.

As far as documentation shows, approximation techniques based on data reduction (e.g. samples and histograms) are not implemented in STREAM. This is confirmed by the test we perform in Section 6.4.4, where we demonstrate that the tuples received by STREAM equals the number of tuples in the query answer. Consequently, no packets are dropped within STREAM. For many queries (e.g. queries involving the aggregating operators `SUM` and `COUNT`) a drop in relative throughput would lead to query answers that are not correct compared to the network traffic it is measuring. For other queries (e.g. queries involving the aggregating operators `MIN`, `MAX`, and `AVG`) a drop in relative throughput may only have a small effect on the correctness of the query answers. For STREAM, correct query answers under low relative throughput is purely based on luck, because approximation techniques are not implemented. As we see from many of the experiments, the relative throughput levels off at approximately 40 % to 50 % when the network load reaches approximately 4 Mb/s to 5 Mb/s. However, this varies from experiment to experiment, and strongly depends on the number of queries executed concurrently. Nevertheless, if we were able to reduce the amount of data with approximately 60 %, using an approximation technique, the query answers would be fairly accurate for all network loads used in the experiments. However, when recognising connections, such a reduction in number of tuples is far from trivial, because connections cannot be recognised without the presence of certain packets. It is possible to not drop SYN and SYN/ACK packets. However, the ACK packets involved in the third step in the connection-establishing handshake do not differ from other ACK packets. Most of the packets in a TCP session contain acknowledgements due to optimisation efforts within TCP. Thus, it is difficult to separate acknowledgements used in the handshake from other acknowledgements.

A surprising result from the second experiment is that the relation sizes do not have any influence on relative throughput. Actually, relative throughput is highest, though by very small margin, when processing the relation containing most tuples. However, with respect to CPU consumption, STREAM performs better when processing relations of smaller sizes. The similarity of performance when processing these relation sizes may be due to the monitoring and adaptive query-processing infrastructure called StreaMon, which is included in the STREAM system. A short description of this infrastructure is given in Section 4.3.4. The join in the query from Task 3 is based on matching port numbers from the relation with destination port numbers in the input stream. There are only two destination port numbers represented by the tuples in the input stream. Consequently, it is possible for StreaMon to obtain and exploit this knowledge by monitoring the input stream.

We always compare relative throughput and CPU consumption to different network loads. Recall that the network load is based on a requested packet size of 628 bytes. Though the average packet size is somewhat larger, as described in Section 6.4.2, a packet size close to 1500 bytes would lead to fewer packet headers on the network, and in turn fewer tuples entering STREAM. Thus, using our workload, relative throughput may be 100 % up to 3 Mb/s, while it may be 100 % up to 5 Mb/s when we use larger packets.

As described in Section 4.4.3, STREAM writes query results to file. Queries producing many tuples every second, leads to large result files. From time to time, as the buffers containing the results fill up, they must be flushed to disk. This happens more often as the query produce more output. Writing query answers to file may have a significant effect on STREAM's performance, because disk I/O is very expensive.

8. Conclusions

In this section, we start by drawing some conclusions based on the results we achieved from designing continuous network monitoring queries (Section 8.1) and by measuring STREAM’s performance (Section 8.2). Furthermore, we summarise our contributions (Section 8.3) and give some critical assessments (Section 8.4) about the work related to the current thesis. Finally, we present possible directions that future work may take (Section 8.5).

8.1 Query Design

We have designed several queries in order to explore the expressiveness of CQL in the current STREAM prototype. In this section, we list the most important aspects revealed.

- STREAM does not include a database that allows insertions, updates, or deletions.
- Only four data types are supported in STREAM. None of them are implemented to deal with network data types such as IP addresses.
- No network operators such as subnet matching are supported in STREAM.
- STREAM does not support tumbling windows.
- In the current STREAM prototype, there are too many hard-coded constants limiting the expressiveness of CQL.

- As tuples arrive within STREAM, no timestamp attributes are added to the tuples. Consequently, it is not possible to process queries that are occupied with the order of tuples in time.

In addition to these limitations, our work with designing queries has also revealed some aspects with STREAM that are more positive.

- STREAM supports sub-queries through the creation of views. This enriches the expressiveness of continuous queries with STREAM. We use sub-queries in all the queries we have designed.

- The possibility of joining streams with relations stored in files provides many opportunities.

- STREAM provides an acceptable set of operators such as the traditional arithmetic operators, average operators, and the bag operators `UNION` and `EXCEPT`.

- Even though join is a binary operator, STREAM allows three or more streams or relations to join sequentially within one expression.

- The possibility to choose if the output of a query should be a stream or a relation is extremely advantageous. In addition, the three relation-to-stream operators enrich the expressiveness of continuous network monitoring queries all the more.

Based on the experiences we have acquired while designing network monitoring queries, we conclude that the expressiveness provided by the CQL implementation in STREAM is relatively rich. It is possible to express a wide range of network monitoring queries. However, some limitations exclude certain queries and force us to express other queries in cumbersome ways.

8.2 Performance Evaluation

Related to simple queries, STREAM has a relative throughput of 100 % up to approximately 30 Mb/s, while the relative throughput is 100 % up to approximately 3 Mb/s for more complex queries. Additionally, we have seen that CPU consumption increases as complexity increases. Concurrently executed queries reveal that relative throughput decreases and CPU consumption increases as number of concurrent queries and query complexity increases. All experiments investigating concurrency includes a collection of relatively complex queries. One of these collections has a relative throughput of 100 % up to 2.5 Mb/s. The relative throughput of the other collections started decreasing at a lower network load.

In some experiments, we accomplished successful investigations of how well STREAM optimises queries. The results reveal that it for many queries is irrelevant whether the user optimises queries prior to executing them in STREAM. This implies that STREAM has good optimisation techniques. However, one of our experiments suggested that when queries have common sub-expressions that are large and complex the relative throughput increases and the CPU consumption decreases when the user optimises the queries prior to execution.

Even though good optimisation techniques seem to be implemented in STREAM, we conclude that STREAM may be used as a network monitoring tool only in very limited environments. We base this conclusion on the results from the experiments, which show that STREAM cannot process continuous network monitoring queries over the high transfer rates revealed in today’s networks.

8.3 Contributions

8.3.1 Query Design

In order to recognise limitations and possibilities introduced by STREAM and its continuous query language, CQL, queries solving a wide range of network monitoring tasks were designed. We selected a collection of tasks that would challenge STREAM as a network monitoring tool. Some of the tasks were relatively straightforward to solve with continuous queries in CQL. Among the not so straightforward tasks, our main occupation was to express queries related to TCP connections. Challenges in expressing these queries included recognising connection establishment, connection lifetime, connection closure, and measuring load throughout the lifetime of connections. We managed to recognise established connections based on the three-way handshake included in TCP. We did not manage to recognise connection closure, because this involves many phases and connections may be closed without accomplishing these phases. By changing the connection recognising technique slightly, we managed to identify SYN packets for which a SYN/ACK packet was sent, but no ACK packet received within two minutes. This technique may be used to recognise SYN flood attacks.

8.3.2 System Implementation

By adding functionality to gen_client, we have made it possible for STREAM to process live network traffic online. In addition, we have implemented a packet capturing system that capture packets, extracts header field values from TCP/IP packets, creates a tuple based on these values, and sends these tuples to STREAM. We have also implemented an experimental setup that allows measurements of STREAMs performance. This experimental setup includes several scripts allowing the execution of a number of experiments in an automatic manner.

8.3.3 Performance Evaluation

We designed several experiments that we used to measure STREAM's performance and investigate aspects within STREAM. The specific impact query complexity and relation size have on STREAM's performance was investigated. Furthermore, we created some experiments that examined the importance of optimising queries by hand, prior to executing them in STREAM. Finally, the performance of STREAM when processing script files of different complexity and number of queries was examined.

8.4 Critical Assessment

Our initial work was related to the first STREAM prototype released, and we soon implemented pushing of live network data into it. However, after some months, a new STREAM prototype was released. We decided to use the newest prototype in our thesis, because it included bug fixes and new operators. Consequently, we had to get familiar with this release and re-implement the live data streaming, because the external interface was altered in the newest prototype.

We have used the term "relatively straightforward" to describe the implementation of some continuous queries. This is relative to the most complex queries we have designed. All queries are designed through several iterations. As our understanding of the continuous query semantics have increased we have realised that earlier solutions are incorrect. Our experience is that semantics of continuous queries are difficult, and that it is a gradual process to gain a sufficient understanding of the subject.

The network traffic produced when we use TG as a traffic generator consists of packets of varying sizes. The reason for this is that TG does not turn of the TCP Nagel algorithm, which delays messages in order to assemble several messages in one packet. This algorithm may be deactivated setting the `NO_DELAY` flag. In addition, the TG server consumes a considerable amount of CPU resources during a couple of seconds approximately every 120 seconds. Over time, we consider this consumption insignificant. We understand that we should have implemented our own traffic generator in order to avoid these problems. However, this was never a subject, because of time constraints related to the project.

Compared to the real world, we have executed the experiments in a restricted environment. The reason for this is that we want to execute all experiments in as equal as possible environments. We have selected parameters as neutral as possible, because we may easily produce an environment that would have resulted in “better” or “worse” results for STREAM. Two important parameters in this matter are number of attributes and packet size. Many of the attributes in the input stream are never queried and by reducing the number of attributes, we would reduce the load of data into STREAM. In addition, using a large packet size would lead to fewer packet headers on the network and in turn fewer tuples in STREAM.

8.5 Future Work

The work we have done in this thesis is merely a beginning within this field of research. Consequently, research may take many possible future directions. In this section, we describe some of the directions this research may take.

Performance Improvement

To become more applicable as a network monitoring tool, STREAM needs to have a relative throughput of 100 % at much higher network loads. Several steps may be taken to achieve this.

- It is possible to reduce the amount of data pushed into STREAM by only including the most relevant header fields in the tuples. It may also be possible to implement some kind of filtering at the NIC.

- Some queries produce an enormous amount of output. For example, queries that use `Rstream` and the `GROUP BY` clause over large windows may cause a high output every second, especially if the stream contains many groups. If the results of these queries are written to file, this may lead to much disk I/O, which is extremely expensive. Consequently, performance may be improved if query results are streamed to e.g. a program that displays the results for instance graphically instead of writing the query results to file.

- The functionality we have implemented in Fyaf may be integrated in gen_client to capture packets directly into the system. This would remove the TCP connection between Fyaf and STREAM.

- We have not tried to find the best configuration of the system by tuning the configuration file. STREAM's performance may be improved by doing this.

Tumbling Windows

Another factor that probably would improve performance is the implementation of tumbling windows. In Section 7.5, we discussed what impact window sizes might have with respect to query complexity. By comparing the windows used in Task 1 and Task 3 we illustrated how many tuples these windows might contain. If STREAM supported tumbling windows, we might reduce the number of tuples in Task 3's window substantially by, firstly, aggregating tuples in small windows and, secondly, in a window, with a size corresponding to the task, aggregating the tuples produced by the first window. In terms of optimizing, this would be like "pushing aggregations" down the query tree. However, this does not apply to the operator `AVG`. In order to calculate average, the operators `COUNT` and `SUM` must be utilised. In order to count the tuples, the small window should use the `COUNT` operator and the larger window should use the `SUM` operator. The three other aggregating operators may be used in both windows. In queries containing a `GROUP BY` clause, the gain in performance much depends on how many groups are represented in the stream. With sliding windows, pushing aggregations into smaller windows would lead to incorrect results, because of the redundancy introduced by this window type. In sliding windows, each tuple contributes to the results several times.

Approximation

In the current STREAM prototype, no approximation techniques are implemented. To preserve a certain correctness of query results as system resources are exceeded, it is necessary with techniques that drop packets or tuples in a controlled manner. In general, approximation techniques reduce data by using summary structures. Examples of such techniques are samples, histograms, and wavelets. An appropriate approximation technique should be implemented in STREAM.

Distributed Stream Processing

In the current thesis, we have considered a DSMS where all data collection and processing take place in a single computer. In many applications, the stream data is actually produced at distributed sources and sent to a centralised DSMS. Examples of such data are the router forwarding tables and configuration data produced at routers. Moving some processing to the sources instead of moving data to a central system may lead to a more efficient use of processing and network resources.

Bibliography

Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: a New Model and Architecture for Data Stream Management. *International Journal on Very Large Data Bases* 12(2), pp. 129-139, August 2003.

Anonymous. Stream Query Repository: Network Traffic Management, 2002. *http://www-db.stanford.edu/stream/sqr/netmon.html*.

Anonymous. STREAM: The Stanford Stream Data Manager. User Guide and Design Document, 2004. *http://www-db.stanford.edu/stream/code/user.pdf*.

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. STREAM: The Stanford Data Stream Management System. Technical Report, 2004a.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., and Widom, J. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin* 26(1), pp. 19-26, March 2003a.

Arasu, A., Babu, S., and Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report, Stanford University, 2003b.

Arasu, A. and Widom, J. A Denotational Semantics for Continuous Queries Over Streams and Relations. *SIGMOD Record* 33(3), pp. 6-11, September 2004b.

Avnur, R. and Hellerstein, J. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, May 2000.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-*

SIGART Symposium on Principles of Database Systems (PODS 2002), Madison, Wisconsin, USA, June 2002.

Babu, S., Subramanian, L., and Widom, J. A Data Stream Management System for Network Traffic Management. In *Proceedings of the 2001 Workshop on Network-Related Data Management (NRDM 2001)*, Santa Barbara, California, USA, May 2001.

Babu, S. and Widom, J. StreaMon: An Adaptive Engine for Stream Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France, June 2004.

Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA, June 2005.

Case, J., Fedor, M., Schoffstall, M., and Davin, J. RFC 1157 - Simple Network Management Protocol (SNMP), 1990. *http://www.faqs.org/rfcs/rfc1157.html*.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Database Research (CIDR 2003)*, Asilomar, California, January 2003.

Chen, J., DeWitt, D., Tian, F., and Wang, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, May 2000.

Chong, C.-Y. and Kumar, S. Sensor Networks: Evolution, Opportunities, and Challenges. *IEEE Special Issue on Sensor Networks and Applications* 91(8), pp. 1247-1256, August 2003.

Clausen, T. and Jacquet, P. RFC 3626 - Optimized Link State Routing Protocol (OLSR), 2003. *http://www.faqs.org/rfcs/rfc3626.html*.

Corson, S. and Macker, J. RFC 2501 - Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, 1999. *http://www.faqs.org/rfcs/rfc2501.html*.

Cortes, C., Fisher, K., Pregibon, D., Rodgers, A., and Smith, F. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, Massachusetts, USA, August 2000.

Cranor, C. D., Gao, Y., Johnson, T., Shkapenyuk, V., and Spatscheck, O. Gigascope: High Performance Network Monitoring with an SQL Interface. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data / Principles of Database Systems*, Madison, Wisconsin, USA, June 2002.

Culler, D. E., Hill, J., Buonadonna, P., Szewczyk, R., and Woo, A. A Network-Centric Approach to Embedded Software for Tiny Devices. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, Tahoe City, California, USA, October 2001.

Eisenberg, A. and Melton, J. SQL:1999, Formerly Known as SQL3. *SIGMOD Record* 28(4), pp. 131-138, March 1999.

Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book*. 1st ed. 1119 pages. Prentice Hall, Inc, Upper Saddle River, New Jersey, 2002.

Gehrke, J., Korn, F., and Srivastava, D. On Computing Correlated Aggregates over Continual Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA, May 2001.

Gehrke, J. and Madden, S. Query Processing in Sensor Networks. *IEEE Pervasive Computing* 3(1), pp. 46-55, January-March 2004.

Geraci, A., Katki, F., McMonegal, L., Meyer, B., and Porteous, H. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. 92 ed. 217 pages. IEEE, 1991.

Goebel, V. and Plagemann, T. Data Stream Management Systems - a Technology for Network Monitoring and Traffic Analysis? In *Proceedings of the 8th International Conference on Telecommunications (ICT 2005)*, Zagreb, Croatia, June 2005.

Golab, L. and Özsu, M. T. Issues in Data Stream Management. *SIGMOD Record* 32(2), pp. 5-14, June 2003.

Grossglauser, M. and Rexford, J. *Passive Traffic Management for IP Operations*. The Internet as a Large-Scale Complex System (K. Park and W. Willinger eds.), Oxford University Press, 2002.

Hussain, A., Bartlett, G., Pryadkin, Y., Heidemann, J., Papadopoulos, C., and Bannister, J. Experiences with a Continuous Network Tracing Infrastructure. In *Proceeding of the ACM SIGCOMM 2005 Workshop on Mining Network Data (Mine-Net 2005)*, Philadelphia, Pennsylvania, USA, August 2005.

Jacobson, V., Braden, R., and Borman, D. RFC 1323 - TCP Extensions for High Performance, 1992. http://www.faqs.org/rfcs/rfc1323.html.

Jain, R. *The Art of Computer Systems Performance Analysis*. 1st ed. 685 pages. John Wiley & Sons, Inc., New York, 1991.

Johnson, T. and Chatziantoniou, D. Extending Complex Ad Hoc OLAP. In *Proceeding of the Eight ACM International Conference on Information and Knowledge Management (CIKM 1999)*, Kansas City, Missouri, USA, November 1999.

Johnson, T., Muthukrishnan, S., Spatscheck, O., and Srivastava, D. Streams, Security and Scalability. In *Proceeding of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2005)*, Storrs, Connecticut, USA, August 2005.

Krämer, J. and Seeger, B. A Temporal Foundation for Continuous Queries over Data Streams. In *Proceedings of the Eleventh International Conference on Management of Data (COMAD 2005)*, Goa, India, January 2005.

Madden, S. and Franklin, M. J. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, California, USA, February 2002.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems* 30(1), pp. 122-173, 2005.

McKenney, P. E., Lee, D. Y., and Denny, B. A. Traffic Generator Software Release Notes, 2002. http://www.postel.org/tg/tg2002.pdf.

Montenegro, G., Dawkins, S., Kojo, M., Magret, V., and Vaidya, N. RFC 2757 - Long Thin Networks, 2000. http://www.faqs.org/rfcs/rfc2757.html.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, California, USA, January 2003.

Plagemann, T., Goebel, V., Bergamini, A., Tolu, G., Urvoy-Keller, G., and Biersack, E. W. Using Data Stream Management Systems for Traffic Analysis - a Case Study. In *Proceedings of the 5th International Workshop on Passive and Active Network Measurement (PAM 2004)*, Antibes Juan les Pins, France, April 2004.

Plummer, D. C. RFC 826 - Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware, 1982. *http://www.faqs.org/rfcs/rfc826.html*.

Postel, J. RFC 791 - Internet Protocol, 1981a. *http://www.faqs.org/rfcs/rfc791.html*.

Postel, J. RFC 793 - Transmission Control Protocol, 1981b. *http://www.faqs.org/rfcs/rfc793.html*.

Siekkinen, M. Measuring the Internet: State of the Art and Challanges, Guest Lecture, inf5090, Spring 2006, Department of Informatics, University of Oslo, 2006.

Spognardi, A., Lucarelli, A., and Pietro, R. D. A Methology for P2P File-Sharing Traffic Detection. In *Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P)*, San Diego, California, USA, July 2005.

Srivastava, U. and Widom, J. Flexible Time Management in Data Stream Systems. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2004)*, Paris, France, June 2004.

Søberg, J. Design, Implementation, and Evaluation of Network Monitoring Tasks with the TelegraphCQ Data Stream Management System. *Master's thesis*, Department of Informatics, University of Oslo, Oslo. Feb. 2006.

Tanenbaum, A. S. *Computer Networks*. 4th ed. 891 pages. Prentice Hall PTR, Upper Saddle River, NJ, 2003.

Terry, D., Goldberg, D., Nichols, D., and Oki, B. Continuous Queries over Append-only Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 1992.

Yao, Y. and Gehrke, J. E. Query Processing for Sensor Networks. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, California, USA, January 2003.

Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., and Balakrishnan, H. The Aurora and Medusa Projects. In *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, March 2003.

Zhu, Y. and Shasha, D. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, August 2002.

Appendix A

Acronyms

ADC	Analogue to Digital Converter
ARP	Address Resolution Protocol
AS	Autonom System
CPU	Central Processing Unit
CQ	Continuous Query
CQL	Continuous Query Language
DBMS	Database Management System
DoS	Denial-of-Service
DSMS	Data Stream Management System
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IPv4	Internet Protocol, Version 4
IPv6	Internet Protocol, Version 6
ISP	Internet Service Provider
LAN	Local Area Network
MANET	Mobile Ad Hoc Networking
MTU	Maximum Transmission Unit

NIC	Network Interface Card
OLSR	Optimised Link State Protocol
P2P	Peer-to-Peer
RAM	Random Access Memory
RTO	Retransmission Timeout
RTT	Round Trip Time
SNMP	Simple Network Management Protocol
TCP	Transport Control Protocol
UDP	User Datagram Protocol
WAP	Wireless Application Protocol

Appendix B

DVD

Enclosed to the thesis is a DVD that contain such as scripts, experiment results, preliminary tests, and offline tests. A graphical illustration of the content structure on the DVD is presented in Figure 38 below. The different directories have content as described in the following.

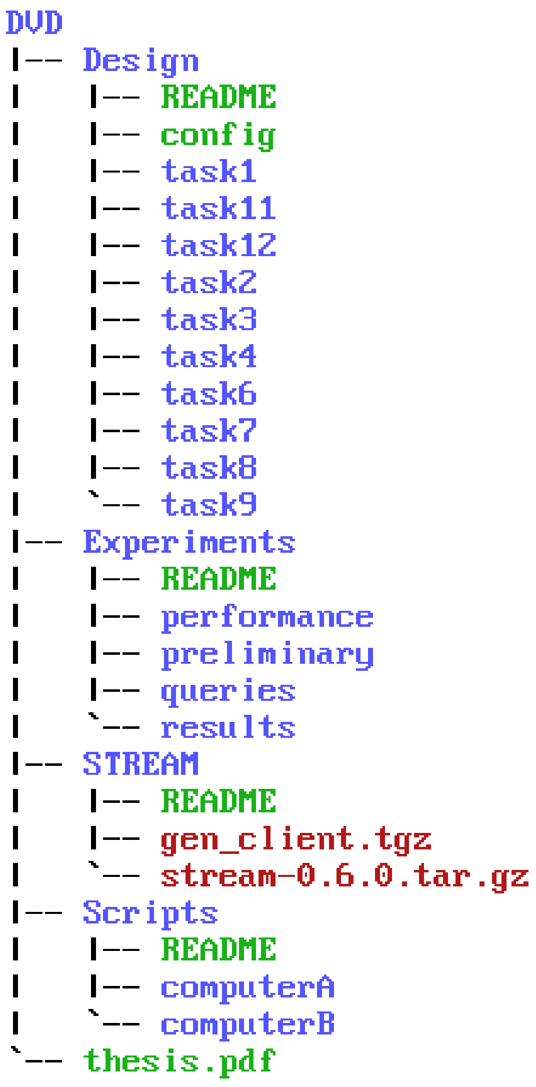
- **Design** contains one sub-directory for each of the queries designed in Section 5.2, except for the tasks in Section 5.2.5 and Section 5.2.10, which was not possible to solve with continuous queries in STREAM. Each sub-directory contains results from the offline tests that confirmed the correctness of the queries.

- **Experiments** contains four sub-directories; performance, preliminary, queries, and results. These the contents of the sub-directories are described in the following. The performance directory contains such as figures and scripts generating these figures for all the six experiments described in Section 7.4. The preliminary directory contains for instance scripts for the tests performed prior to the experiments. These tests are described in Section 6.4. The queries directory contains script files describing the queries used in the six experiments. The results directory contains results from all experiments.

- **STREAM** contains one tar-file for STREAM-0.6.0 and one tar-file with the changes done to gen_client in order to allow a live data source.

- **Scripts** contains two sub-directories; computerA and computerB. These directories contains the different scripts described in Section 6.3.3.

In addition to these the directories described above, we have enclosed the a PDF file containing the thesis.



```
DVD
|-- Design
|   |-- README
|   |-- config
|   |-- task1
|   |-- task11
|   |-- task12
|   |-- task2
|   |-- task3
|   |-- task4
|   |-- task6
|   |-- task7
|   |-- task8
|   `-- task9
|-- Experiments
|   |-- README
|   |-- performance
|   |-- preliminary
|   |-- queries
|   `-- results
|-- STREAM
|   |-- README
|   |-- gen_client.tgz
|   `-- stream-0.6.0.tar.gz
|-- Scripts
|   |-- README
|   |-- computerA
|   `-- computerB
`-- thesis.pdf
```

Figure 38. Content structure of the DVD enclosed to the thesis