# On-Demand View Materialization and Indexing for Network Forensic Analysis

Roxana Geambasu[1], Tanya Bragin[1], Jaeyeon Jung[2], and Magdalena Balazinska[1]

[1] Department of Computer Science and Engineering
University of Washington, Seattle, WA
{roxana,tbragin,magda}@cs.washington.edu

[2] Mazu Networks
Cambridge, MA
jyjung@mazunetworks.com

## Abstract

*Today, network intrusion detection systems (NIDSs) use custom solutions to log historical network flows and support forensic analysis by network administrators. These solutions are expensive, inefficient, and lack flexibility. In this paper, we investigate database support for interactive network forensic analysis. We show that an "out-of-the-box" relational database management system (RDBMS) can support moderate flow rates in a manner that ensures high query performance. To enable support for significantly higher data rates, we propose a technique based on on-demand view materialization and indexing. In our approach, when an event occurs, the system proactively extracts relevant historical data and indexes it in preparation for forensic queries over that data. We show that our approach significantly improves response times for a large class of queries, while maintaining high insert throughput.*

## 1 Introduction

Traditionally, a network intrusion detection system (NIDS) monitors traffic at a network *perimeter* to block known attacks and suspicious network behavior (e.g., port scans). However, as attackers devise new ways to compromise and exploit end hosts (e.g., email viruses, peer-to-peer malware), many enterprises have begun to employ more sensors *inside* their network to monitor internal network activity. As such, an NIDS has evolved to support internal network security—e.g., detecting internal worm propagation.

The main role of an NIDS is to detect and flag suspicious network activity in near real time. Some alerts can trigger an automatic response such as dropping malicious packets or updating a network filter. Many alerts, however, go through a manual verification process in order to sift out false alarms. This verification process typically involves checking the suspected host's recent network activity and looking up any services recently run by that host. It may also involve forensic investigation aimed at finding the root cause of the security breach and identifying other affected hosts. To support the network administrator, an NIDS must thus provide not only near real time security event detection, but also convenient and efficient access to historical network flow data. Near real-time network event detection has received significant attention in the past [9, 18, 20]. In this paper, we address the problem of building a historical flow database suitable for forensic analysis queries. Figure 1 shows the structure of an NIDS with a historical log database.

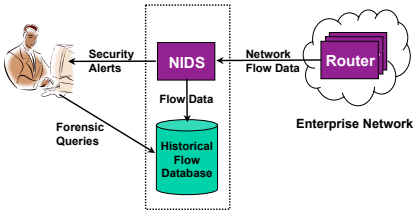


Figure 1: **Network intrusion detection system with a historical flow database.** The NIDS receives network flow summary data (such as NetFlow data [8]) from the enterprise's internal routers, processes them, and generates alerts. While doing so, it maintains historical flow summary data for network forensic analysis.

Supporting a historical network flow database in conjunction with an NIDS raises two important technical challenges. First, because network traffic monitors generate data continuously and at high-rate, the database needs to support a high data insertion rate. For example, in a busy hour, a medium-size enterprise network with several thousands of hosts may see up to 100,000 new flows per minute internally. Second, to facilitate human-driven forensic analysis, the database must answer historical queries quickly ($<$ seconds). Today, NIDSs try to meet these challenges by implementing their own storage systems; such custom solutions, however, are expensive to build and often offer limited and sometimes awkward APIs [13].

In contrast, relational database management systems (RDBMSs) [12, 14, 17, 19] offer many features desirable for conducting network forensic investigations—a pow-

erful query optimizer, indexes, and a flexible and standard query language (e.g., SQL). It is common belief, however, that an "out-of-the-box" RDBMS is ill-suited to support the high data-rate workload of an NIDS. Even if an RDBMS can store incoming data sufficiently fast to keep-up with input data rates (by bulk loading the data, for example), efficient querying requires indexes on multiple attributes, and indexes are known to significantly decrease database insert throughput [5]. Hence, it is expected that an RDBMS would be either too slow to keep-up with network flow data rates or it would have unacceptably slow query execution times.

In this paper, we first demonstrate this trade-off between insert throughput and query performance for an off-the-shelf RDBMS using a network monitoring and forensic analysis workload (Section 2). We show that an RDBMS has excellent insert throughput as long as it only keeps one or two indexes on the data (23,000 flows/sec for two indexes in our experiments), but that such a small number of indexes yields an unacceptably slow response time for common forensic queries (from a few minutes to several hours).

To address this shortcoming, we develop a new technique that we call OVMI, for *On-demand View Materialization and Indexing* (Section 3). The OVMI technique enables an NIDS to use an off-the-shelf RDBMS as the network flow archive, yet ensure *both* a high data insert rate and a high forensic query performance. OVMI exploits three properties of forensic analysis queries to achieve this goal. First, alerts produced by the NIDS partly determine subsequent forensic queries: i.e., administrators query the system about the historical activity of the *entities involved in the newly detected event*. Second, forensic queries are "drill-down" queries: subsequent queries refine the selection predicates of earlier queries. Finally, having a human in the loop creates a time delay between the time when an event occurs and the time when a first forensic query is issued. OVMI exploits this delay to prepare for upcoming queries.

With OVMI, the RDBMS does not index data continuously. Instead, when an event occurs, the NIDS requests that the RDBMS heavily index *only* the recent network activity of the entities involved in the event. To ensure fast creation of these partial indexes, the RDBMS first copies all relevant data into a separate table and builds indexes on that table. The existence of these materialized views is known to the NIDS but transparent to the user. We show that the OVMI approach works well for a large class of forensic analysis queries, where an administrator starts submitting queries within a short time period (minutes to tens of minutes) of an event and is interested in the past few hours worth of network flow data

OVMI improves query evaluation performance by orders of magnitudes while supporting high data rates (up to 3,000 flows/s in our experiments). OVMI also helps quickly produce partial results when a query stretches past the materialized time window of data.

## 2 Storing Flow Records in an RDBMS

The data rate that an NIDS must handle depends on several factors, including the number of hosts on the monitored network, the applications that these hosts are running, and hosts' network activity levels. Roughly, we estimate that a flow *arrival* rate can easily reach 100,000 flows per minute ($\approx$ 1,700 flows/s) for a medium-large enterprise where thousands of active hosts are generating traffic[1]. The flow arrival rate can be even higher for larger organizations or if a network worm causes infected machines to engage in network scans.

In this paper, we assume that an NIDS receives streams of flow records from network sensors or routers, which track each flow, updating the flow's attributes such as the number of packets, bytes, and TCP flags. These sensors or routers can be configured to export a set of flow records to an NIDS for further processing when these flows expire. Our goal is thus to devise a historical flow database that will easily handle flow record insert rates in excess of 3,000 flows/s. In this section, we benchmark the performance of an RDBMS in handling such data rates while maintaining various indexes in order to provide good query performance.

### 2.1 Method

We perform all experiments on a PostgreSQL 8.1.5 database [19]. PostgreSQL is one of the most advanced open source databases. We expect the performance of commercial databases, however, to exceed that of PostgreSQL.

For our experiments, we use two traces: *Trace 1* is a 10-hour network trace from a medium-size Internet Service Provider (ISP). This trace is somewhat old, collected in April 2003, but contains flow records from two hosts infected by a Code Red worm (out of 389 hosts), scanning hundreds of thousands of IP addresses, making it suitable for testing RDBMS performance in the presence of security events. *Trace 2* is a 22-day network trace from a small enterprise collected in October—November 2006. Table 1 shows the main attributes associated with the network flow data in both traces. The average flow rates are 27 flows/s for Trace 1 and 10 flows/s for Trace 2. Since our traces are over limited periods of time, we cycle them to simulate more extended monitoring activity. We also replay them at higher speeds to simulate higher data rates.

[1] We have seen on average 50,000 new flows per minute from a network with about 400 hosts in business hours.

| Column | Type | Description |
|---|---|---|
| start_ts | timestamp | start time of flow |
| protocol | varchar(10) | TCP, UDP, ICMP, etc. |
| cli_ip | int | client IP |
| srv_ip | int | server IP |
| cli_port | varchar(8) | client port |
| srv_port | varchar(8) | server port |
| app | varchar(20) | application |
| c2s_p | int(64) | # packets from client to server |
| s2c_p | int(64) | # packets from server to client |

Table 1: **Schema of a network flow database**. We show only attributes relevant to our discussion. All attributes can be obtained from network packet headers, except `app` (e.g. MSExchange, Lotus Notes), which is available via payload inspection. Flows are generated by `app`.

For all benchmarks, we use a machine with two Intel Xeon, 3 GHz processors with hyper-threading enabled, 8 GB RAM, and a 369 GB, Western Digital (SATA, 8.9ms seek, 16Mb cache, 7200 RPM) disk. We believe that this machine is representative of what one would use for a DBMS server.

## 2.2 Database Bulk Load Throughput

The database insert throughput determines the maximum average flow rate that the database can keep up with. Figure 2 shows the insert throughput for increasing numbers of indexes. In this experiment, the client bulk-loads the data, blocking between each bulk. The figure shows results for 500-tuple bulk sizes. Error bars show maximum throughputs as the bulk size changes. The indexed attributes are in order: `start_ts`, `cli_ip`, `srv_ip`, `proto`, `srv_port`, `cli_port`, and `app`. All indexes are B-trees to enable range queries.

As shown in the figure, without indexes, the database can sustain rates in excess of 45,000 flows/s. Each added index significantly cuts the throughput. For Trace 2, even though the throughput decreases, the database can sustain all 7 indexes and still achieve our 3,000 flows/s goal. However, this is not the case in general. For Trace 1, the throughput falls below 3,000 flows/s after 5 indexes. Additionally, the throughput is barely above 3,000 flows/s as soon as the database maintains more than two indexes. Therefore, maintaining multiple indexes is not a scalable approach to achieving high input rates.

Figure 2 also suggests that the effect of an additional index on throughput depends on the properties of the indexed attribute. Figure 3 shows this effect in more detail by presenting the maximum insert throughput for Trace 1 with a single index but for different indexed attributes. Attribute properties that affect performance include the data distribution: e.g., `start_ts` is clustered and monotonically increasing, while (`srv_ip(int)`) is unclustered and has a large number of distinct values due to port scans by the infected hosts. Another important property is the attribute's *data type*: e.g., indexing integers (`srv_ip(int)`) goes significantly faster than indexing strings (`srv_ip`). Figure 3 also shows that the database can maintain almost the same throughput whether it has a single *multi-attribute* index (e.g., `(cli_ip,srv_ip)`) or an equivalent number of *single-attribute* indexes (e.g., `cli_ip` and `srv_ip`). Grouping indexes into multi-attribute indexes thus provides minimal performance gains.

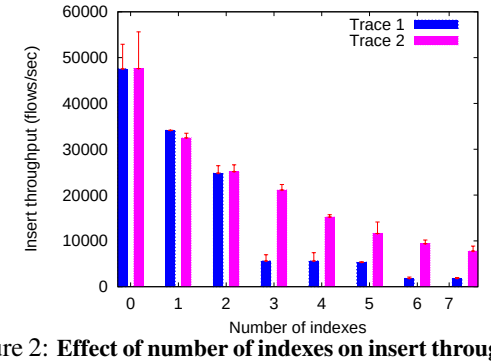


Figure 2: **Effect of number of indexes on insert throughput**. Indexes significantly cut throughput for both traces.

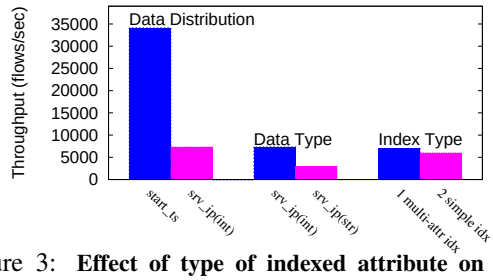


Figure 3: **Effect of type of indexed attribute on insert throughput**. The throughput of a database depends on the properties of the indexed attributes. `srv_ip(int)` represents a hash value of the string `srv_ip(str)` attribute.

Other properties than the ones we observed can also affect performance. However, because the choice of indexes must be based on workload and not index performance, and because the distribution of input data cannot be controlled, we expect the maximum possible number of indexes to remain low in practice.

## 2.3 Network Forensics Queries

Suppose a host `X` is detected as having been scanning hundreds of hosts on TCP port 25 at a slow rate for the past two days. This event may lead the administrator to issue the following queries to identify the root-cause of this scanning behavior (e.g., email virus propagation or unregistered email server):

**(Q1) What client applications has `X` been running and how much traffic was generated for each application?**

```
SELECT count(*) FROM Flows
WHERE start_ts >= 'now - T' AND cli_ip = X
GROUP BY app
```

Q1 is one example of forensic analysis query. In general, administrators need to issue several such queries,

using different attributes in their search. For example, to investigate the same alarm, the administrator may further need to see what applications X has been running as a server instead of as a client (i.e., WHERE srv_ip = X) (**Q2**) or both as client and server (i.e., WHERE cli_ip = X OR srv_ip = X) (**Q3**).

After running some initial queries, the administrator often needs to execute additional queries to "drill-down" on the problem. Let us assume that the administrator executed query Q3, which returned the following result:

```
WEB 52563
MSExchange 2231
NETBIOS139 476
. .
```

Given the NETBIOS and MSExchange traffic, the result suggests that the host in question is running a Microsoft Windows operating system. The administrator is now suspicious of a possible infection by a MyDoom email virus. To verify her suspicion, she checks whether X has received any traffic on port 1034 which is known to be a backdoor used by MyDoom.

**(Q4). Has X received any traffic on port 1034?**

```
SELECT s2c_p, c2c_p FROM Flows
WHERE start_ts >= 'now - T' AND
      srv_ip = X AND srv_port = 1034
```

The administrator may also submit more sophisticated SQL queries. An example is to look for backdoor intrusions: the administrator wants all pairs of malicious "triggers", in which a first attack flow causes the victim to initiate a new flow back to the attacker to register the success of the exploit (**Q5**).

```
SELECT * FROM Flows a, Flows b
WHERE a.srv_ip = b.cli_ip AND
      a.cli_ip = b.srv_ip AND
      a.srv_port = 1034   AND
      a.start_ts < b.start_ts AND
      b.start_ts - a.start_ts < 600 -- seconds
```

To support a variety of such queries effectively, an RDBMS needs to maintain a variety of indexes on the different attributes that appear in query predicates. With only one or two indexes, a large fraction of queries will require scanning all historical data within the window $T$, yielding an unacceptably poor performance. For example, for an average flow rate of 3,000 flows/s and with only an index on start_ts, Q2 takes about 19s for $T = 1$ h, about 6 min for $T = 6$ h, about 57 min for $T = 1$ day, and as much as 5.1 h for $T = 2$ days. With an additional index on srv_ip, the same queries take under one second.

Fortunately, forensic queries have several properties that we can leverage to support them efficiently:

1. Forensic queries are highly predictable. By definition, these queries request historical information about the recent network activity of entities involved in alerts. Alert attributes (e.g., alert time, IP address of the machine generating the abnormal network activity) thus determine the bulk of upcoming forensic queries.
2. Forensic queries look at a tiny fraction of all the network flow data. In our traces, we find that even the busiest server is responsible for less than 3% of all flows.
3. Forensic analysis is usually composed of one or more *drill-down* queries—subsequent queries refine the selection predicates of earlier queries, requesting a subset of previously selected tuples.
4. There is typically a time lag between the time an alert occurs and the time an administrator investigates it.

In what follows, we propose a scheme that exploits these properties to efficiently support network forensics queries in an off-the-shelf relational database.

# 3 On-Demand View Materialization and Indexing (OVMI)

To support network forensics queries, we propose to maintain only a full index on start_ts and create additional indexes over the historical flow log *on-demand* (i.e., when an event occurs) and *partially* (i.e., only index data that is relevant to the alert). Since our scheme (OVMI) does not rely on multiple full indexes, the database can sustain a high average insert rate of network flow data. At the same time, because OVMI still indexes the relevant data, it provides fast query execution. Given that forensic queries are predictable (Section 2.3), a simple rule-based deterministic algorithm can be used to select the predicates and indexes for each newly detected event.

## 3.1 Partial Materialization and Indexing

PostgreSQL already supports partial indexes [19], where an index is built only over tuples that match a particular predicate. However, when building a partial index, PostgreSQL does not utilize existing indexes that match the predicate but instead scans *the entire* underlying relation. This approach is untenable in our setting because the historical log is large and grows with time.

Instead, we propose to use a materialized view. When an alert occurs, OVMI reacts by copying into a temporary table all network flow data that falls within some predefined time window (T) and matches a predicate determined by the alert. In doing so, OVMI utilizes the index on start_ts and avoids scanning the whole relation. The following example shows a materialized view created upon scan_event_1.

```
SELECT * INTO scan_event_1 FROM Flows
WHERE start_ts >= 'now - T'  AND
      start_ts <= now AND
      (cli_ip = X OR srv_ip = X)
```

| | Window Size | | | |
|---|---|---|---|---|
| | **1 hour** | **6 hours** | **1 day** | **2 days** |
| **Scan Window** | 19 s | 6.2 min | 56 min | 5 h |
| **Materialize 5%** | 24 s | 6.5 min | 58.4 min | 5.3 h |

Table 2: **Scanning and materializing various time windows.** View materialization is largely dominated by the time it takes to scan the window. We assume a flow rate of 3,000 flows/s and materialize 5% of the window.

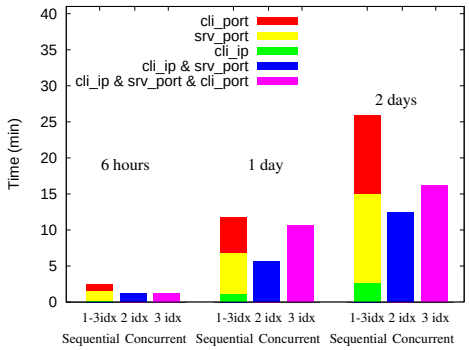


Figure 4: **Time to index the materialized view.** We are showing the time to create one through three indexes concurrently and compare it with the time to create them sequentially. Concurrent indexing is significantly faster than sequential indexing and it is little compared to view materialization.

As mentioned in Section 2.3, typical forensic queries only look at a small percentage of the tuples in the time window (T). Table 2 shows the time it takes to scan a window T and to materialize 5% of the tuples in T, assuming a flow rate of 3,000 flows/s.[2] We see that the view materialization time is largely dominated by the time it takes to scan the window. For example, materializing 5% of a 1-day window incurs only about 4.2% penalty over scanning the window.

5% of a large time window can still represents a significant amount of data if the flow rate is high. For example, six hours of flow data at 3,000 flows/s represents 65 million tuples. While our simple queries from Section 2.3 take on the order of seconds to complete even on the 5% view of a 2-day window (e.g., query Q1 takes about 80 sec), in the absence of indexes, more complex queries can take significantly longer because the database can only access the data by scanning it.

To achieve high query performance, indexes must be built on the materialized view. Figure 4 shows the time to build between one and three indexes simultaneously on the materialized view. For 5% of one day, even three simultaneous indexes take only about 10 min to build. Index creation drops to 1 min for a 6-hour window. Indexing the materialized view thus adds a small overhead compared with materializing the view in the first place.

[2]The results shown are for Trace 2. We found almost identical results when using Trace 1.

Overall then, the bulk of the cost to materialize and index the data in preparation for forensic analysis queries is dominated by the time to scan the historical window of data. The time varies greatly depending on the size of this window. Therefore, OVMI can serve to prepare a shorter or longer window of historical data, depending on flow rates and the expected delay between the time an alert occurs and the time the administrator starts his investigation. With 3,000 flows/s and for a typical scenario where the time lag is on the order of minutes or tens of minutes, the OVMI approach enables the system to easily pre-process over 6 hours worth of data. Additionally, as we discuss in the following subsection, OVMI also benefits queries with a window of interest larger than that of the materialized view.

One limitation of this approach is that potentially important network traffic that follows the initial alert is not included in the view. To address this limitation, we can extend the above approach as follows. When the NIDS triggers the alert, the system materializes and indexes a given window of data. It then continues to insert new data into the materialized view until the administrator indicates that the alert is no longer interesting. While updating the materialized view, the database throughput would decrease. The decrease, however, would be temporary. Once the threat is handled, the materialized view would no longer need to be updated, and the database throughput would go back to its pre-alert value.

## 3.2 Partitioned Query Execution

In some scenarios, forensic queries may request more information than is available in the materialized view. For these scenarios, OVMI is still helpful, because the materialized view that it prepares can help answer a subset of the query faster. There exists a vast literature on exploiting materialized views to improve query execution performance [10, 11]. In our approach, although we support arbitrary queries over the materialized view and the raw archive, we currently only split selection queries on their time predicates, which is enough to demonstrate the benefit of OVMI.

More specifically, given a query that overlaps a materialized view, we split it into two subqueries based on time: one subquery runs on the view exploiting its indexes, and the other one accesses the data via a scan of the remaining time window in the `Flows` relation. For example, given the sample materialized view from Section 3 over the last $T = 6$ h and query Q4 over $T = 7$ h, the query is split into the following subqueries:

```
(Q4.1) SELECT s2c_p, c2c_p FROM scan_event_1
       WHERE srv_ip = X AND srv_port = 1034
(Q4.2) SELECT s2c_p, c2c_p FROM Flows
       WHERE srv_ip = X AND srv_port = 1034
       AND start_ts >= 'now - 7'
       AND start_ts < 'now - 6'
```

| Hrs. Inside Mat. View + Hrs. Outside | Time | |
|---|---|---|
| | Results from Mat. View + Results from Flows | Unsplit Query |
| 5h + 1h | 0.02 s + 21 s | 6.3 min |
| 1h + 5h | 0.02 s + 4.8 min | 6.3 min |

Table 3: **Performance of split queries.** We show the execution time for both subqueries of the split query, for different window sizes covered by or outside the materialized view. Splitting queries improves query performance and allows the user to get the first results fast.

Table 3 shows the performance of partitioned Q4 queries if a materialized view indexed on `srv_port` exists. We vary the overlap of the query's time interval and the view's window. In addition to the virtually zero delay in retrieving the first results from the indexed materialized view, splitting the query also improves its overall performance: the larger the window covered by the materialized view the greater the gain. For example, about 94% gain is achieved in the favorable case of having 5 hours covered by the view and only 1 hour lying outside. In summary, OVMI greatly improves performance not only for queries that match the materialized window of data, but also for queries that span a larger time interval.

However, there are cases when splitting queries might penalize the total execution time. For example, if we only materialize flows with `cli_ip = X` within a certain time window, splitting the query `SELECT * FROM Flows WHERE start_ts > T AND (cli_ip = X OR cli_ip = Y)` will yield worse performance than executing the query only on the `Flows` relation. For this reason, we currently do not perform query splitting for this kind of queries, although they still improve user-perceived latency by returning the first results fast.

## 4 Related Work

Traditional NIDS systems detect intrusions, known attacks, and suspicious network activity [18, 20]. In this paper, we focus on a newer class of NIDS systems that provide both intrusion detection functionality and network forensic analysis by keeping historical flow logs [1, 2]. Existing NIDS solutions use custom databases whose performance results are not publicly available. We show that it is possible to use an open source relational database to store network flow information and propose an on-demand materialization and indexing technique that effectively speeds-up forensic queries, while maintaining a high insert throughput.

The increased popularity of monitoring applications, including network monitoring and network intrusion detection, has lead the database community to develop a new class of general-purpose data management systems, called stream processing engines (SPEs)[3]. Examples of SPEs include: Aurora [3], Borealis [4] Gigascope [9], STREAM [15], and TelegraphCQ [6]. These SPEs process flow data directly as it arrives without storing it. As such, they provide high performance continuous processing, but do not support forensic analysis queries. Our approach is thus complementary to these efforts.

The Seaweed [16] peer-to-peer data management system can serve as a historical flow database. Seaweed's focus, however, is on the availability and scalability challenges of accessing highly distributed data sets. It provides no mechanism for speeding-up historical queries at each site, which is the focus of our approach.

Answering queries using materialized views [10, 11] and caching query results [23] are known techniques to speed-up query execution. Our contribution lies in applying these techniques to the domain of network forensic analysis and proposing a new scheme that materializes relevant network flow data *on-demand*, when an event occurs, to speed-up upcoming queries.

Partial indexes, which enable a database to index only tuples matching a predicate, have been investigated in the past [21, 22, 24]. Our approach can be thought of as building partial indexes on-demand. However, instead of using the native partial index mechanism, we materialize data matching the predicate to avoid the overhead of scanning the entire relation while building the index.

FELIX [5] is an approach for temporarily suspending indexing archived stream data during overload. In contrast, in our application domain, the database is unable to maintain full indexes under normal load conditions.

Handling large amounts of historical data is not a new problem in databases. However, most research in this area has focused on warehousing scenarios, where data is archived, indexed and queried offline, with plenty of time to optimize reads for data warehousing applications [7].

## 5 Conclusion

Forensic analysis is a critical task in network monitoring applications. In this paper, we proposed OVMI, an On-demand View Materialization and Indexing technique, that enables an NIDS to use an off-the-shelf RDBMS for its historical log. With OVMI, the system can sustain high network flow rates, while achieving high performance for forensic analysis queries. We showed that with a data rate of 3,000 flows/s, OVMI can prepare the system within minutes to handle forensic analysis queries over several hours of data preceding the alert. We view OVMI as an important step in the direction of using RDBMSs instead of custom solutions in network monitoring applications.

[3]These engines are also called data stream management systems (DSMS) [3, 15] continuous query processors [6], and stream databases [9]

One limitation of OVMI is that it improves performance for forensic analysis queries over only the most recent several hours worth of historical data. While this amount of data is frequently sufficient, administrators sometimes need to access even older data. In future work, we plan to investigate additional techniques to further improve RDBMS support for applications within the network monitoring domain, including efficiently querying larger volumes of historical data. We also plan to investigate archiving and efficiently querying data other than just flow data.

## References


[1] Mazu networks. http://www.mazunetworks.com/, 2007.

[2] Netscout. http://www.netscout.com/, 2007.

[3] Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.

[4] Abadi et. al. The design of the Borealis stream processing engine. In *Proc. of the CIDR Conf.*, Jan. 2005.

[5] S. Chandrasekaran. Query processing over live and archived data streams. Ph.D. Thesis, Univ. of California Berkley, 2005.

[6] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the CIDR Conf.*, Jan. 2003.

[7] S. Chaudhuri and U. Dayal. Overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1), 1997.

[8] Cisco Systems. Cisco IOS NetFlow. http://www.cisco.com/go/netflow, 2007.

[9] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 SIGMOD Conf.*, June 2003.

[10] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. of the 2001 SIGMOD Conf.*, 2001.

[11] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[12] IBM. DB2. http://www.ibm.com/db2, 2006.

[13] D. N. Joel Snyder and R. Thayer. http://www.networkworld.com/reviews/2003/1013idsrev.html?page=3, 2003.

[14] Microsoft. SQL Server. http://www.microsoft.com/sql/, 2006.

[15] Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the CIDR Conf.*, Jan. 2003.

[16] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with Seaweed. In *Proc. of the 32nd VLDB Conf.*, Sept. 2006.

[17] Oracle. http://www.oracle.com/, 2006.

[18] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[19] PostgreSQL. http://www.postgresql.org/, 2006.

[20] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of the 13th Large Installation System Administration Conference*, pages 229–238, 1999.

[21] C. Sartori and M. R. Scalas. Partial indexing for nonuniform data distributions in relational DBMS's. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):420–429, 1994.

[22] P. Seshadri and A. N. Swami. Generalized partial indexes. In *Proc. of the 11th ICDE Conf.*, 1995.

[23] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *SSDBM '99*, 1999.

[24] M. Stonebraker. The case for partial indexes. Technical Report UCB/ERL M89/17, EECS Department, University of California, Berkeley, 1989.