Data Partitioning and Indexing for Network Forensic Analysis

Cherie Cheung Jue Wang
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{cherie,juewang}@cs.washington.edu

1. BACKGROUND AND MOTIVATION

Nowadays, both Internet service providers and enterprise network administrators need to record and analyze network traffic stream data for network management, diagnosis and security reasons. In their systems, data streams are usually collected continuously at a high rate. Hence, the ability to query promptly on the historical data is highly desirable for the decision making process. A network intrusion detection system (NIDS) is a typical example and the target for this research project. One major functionality of an NIDS is to record network flow data into its associated database system in real-time. The insertion rate is usually considerably high. For example, a medium-size enterprise network with several thousands of hosts can produce 100,000 internal flows per minute at peak time. Moreover, NIDS systems are very sensitive to true negative cases (where a real attack occurs but not detected) and the false-positive rate is relatively high. This requires manual verification, or even forensic analysis with "drill-down" queries aimed at finding the root cause of the security breach and identifying other affected hosts. Both verification and forensic analysis require a prompt response to historical queries in the order of seconds.

To meet these challenges, approaches from three directions have been proposed and explored: a) Customized solutions with their own storage and query systems; such systems, however, are expensive to build and often offer limited and sometimes awkward APIs. b) Data warehousing systems [5, 9] based on relational database management systems (RDBMS); such systems provide independent OLAP (online analytical processing) engines and sophisticated ETL (extraction, transportation, transformation, and loading) solutions. The major problem with data warehousing is the cost of the maintenance of the pre-computed and pre-fetched data (e.g., materialized views). c) An RDBMS itself offers many attractive features for conducting network forensic investigations: a powerful query optimizer, indexes, and a flexible and standard query language interface. However, to make RDBMS suitable for use in this context, it will need to be modified to support high insertion rate and efficient query processing. Previous work [7] has attempted to address the challenges by on-demand materialized view and indexing. We believe RDBMS based solutions can be further improved with data partitioning and smart indexing techniques and ultimately achieve the performance required by network forensic application.

The rest of the paper is organized as follows: In section 2, we discuss related work; In section 3, we present our methodology; In section 4, we present our system design and implementation. In section 5, we evaluate our system performance. In section 6, we conclude and describe possible future work.

2. RELATED WORK

Researchers and practitioners have tried to tackle the problem of providing both high insertion rate in real time and prompt query processing in NIDS from various angles.

In [7], Roxana et al. propose an on demand view materialization and indexing (OVMI) mechanism based on an "out-of-the-box" RDBMS. When an alert comes, OVMI retrieves data involving entities in the alert event, makes a copy of these data and builds rich indexes on the copied data. User queries will be first executed on the materialized data using the indexes and then on the original relation if necessary. With OVMI, the indexing overhead is essentially reduced and the materialized view can assure prompt responses to user queries if the data being queried is within the scope of the materialized view. However, the cost of constructing the materialized view from the full relation is not trivial. And it is very hard to decide the proper time range where the relevant data will be located for complex queries. Fixed heuristics on the time range and data source for the materialized view will lead to limited query flexibility. If user queries need to explore data out of the scope of the materialized view, the overall performance could be worse than a plain system without OVMI.

In [9], Theodore et al describe Data Depot, a general purpose tool for generating warehouses from stream data sources. Data Depot supports efficient query processing by providing complex materialized views over an underlying database. The database is partitioned chronologically such that only selected partitions are affected by updates. Data Depot provides a declarative mechanism for specifying the temporal relationships between partitions of materialized views and their sources with simple formulas. Data Depot also supports complex dependencies and correlations for speeding up query processing. With all these advantages, none the less, the extra cost of maintaining complex materialized views is costly. And the materialized views themselves could be arbitrarily complex and non-monotonic. Data Depot can work in both offline and online environments, while the latter requires high insertion rate and has less resources left for query processing. The cost of the data warehousing solution is formidable for many small or mid-sized corporations or non-profit institutions. In this sense, an RDBMS based approach is desirable.

In [6], a column based relational DBMS is proposed. In

this approach, data are stored by columns rather than by rows. Careful data compression is performed to minimize the disk bandwidth usage. The data is grouped into different column-based projections. The same column may exist in multiple projections, possibly sorted on a different attribute in each of them. This overall design is targeted at optimizing the reading performance. While C-Store will certainly improve the performance of selection queries if the columns and indexes referred by the queries exist in the column projection, this approach has several disadvantages which make it inappropriate for NIDS tasks: 1) It consumes a huge amount of computation resources for data encoding, decoding and sorting. This will limit its application in NIDS where computation resources are scarce. 2) It is optimized for read-only tasks. As NIDS with forensic analysis is naturally write-intensive, and query response needs to be prompt, C-Store is not suitable.

Our work will focus on avoiding the extra cost of maintaining materialized views and providing a solution based on "off-the-shelf" commercial RDBMSs. We will address our methodology in the next section.

3. METHODOLOGY

To improve the query processing and data insertion performance of an "off-the-shelf" relational database for network forensic application, we will explore data partitioning and indexing techniques. The key idea of data partitioning involves subdividing a table into multiple smaller pieces. Each of these partitions will contain a subset of the original data. This is useful when the original table is large and the portion related to each query can be separated from each other by attribute columns or over a particular attribute. For instance, we can partition the network flow database with time as the partition key. If we issue a query over a time period T, we only need to access the partitions with flow data that overlaps with T which saves us time from scanning the whole table. This kind of partitioning also speeds up data insertion. Since newly arrived flow data will only affect the latest partition, after the data insertion, we only need to rebuild the index over that partition instead of the whole table. The challenge will be to find a good way to partition the table so that the partitioning overhead is relatively small comparing with the performance gain and the partition size suits most queries well. Indexing is another technique for optimizing query performance by enabling rapid random lookup. Depending on the kinds of indexes and the attributes on which the indices are built, indexing can speed up different kinds of queries.
In particular, we are interested in optimizing drill-down queries and recursive queries that often appear in network forensic analysis. However, due to time constraint, we haven't investigated specific strategy for optimizing them in this work. Yet our experiments show that partitioning with indexing are very effective for optimizing the query performance for general queries including "drill-down" and recursive queries.

4. IMPLEMENTATION

In this section, we will outline our system setup and the properties of our data traces, followed by the implementation details of each of our system modules: data insertion, indexing and partitioning.

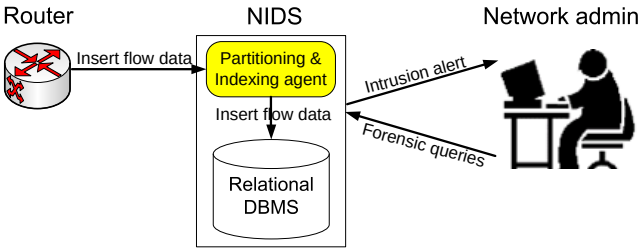

Figure 1: Our system architecture

4.1 System setup

An NIDS system consists of a historical flow database that stores all the network flow data from a router. When an attack is detected by the NIDS, it will send an alert to the network administrator, who then issue forensic queries to verify and find the origin of the attack. We proposed to use an "off-the-shelf" RDBMS as the historical flow database and implemented a partitioning and indexing agent on top of that to enable fast data insertion and low query processing time. Figure 1 shows our system architecture.

The Microsoft SQL Server 2005[2] was used as our database backend. The partitioning and indexing agent was implemented in Java. It would connect to the database using JDBC[8] and issue SQL commands to it for performing data insertion, indexing and partitioning operations. These operations were interleaved so that data was inserted in real-time and queries were processed promptly. The SQL Server run on the Microsoft Windows Server 2003[3] on a machine with 3.00GHz Intel Xeon qualcore CPU, 4.00GB of RAM and one 750GB SCSI harddisk.

4.2 Analysis of the data traces

For our experiments, we used two traces: Trace 1 is a 10-hour network trace from a medium-size Internet Service Provider (ISP). This trace was collected in April 2003. It 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. In order to achieve the best performance, we converted and represented the attributes of the network flow data using as little memory as possible. Table 1 shows the schema of the main attributes of our database.

Table 1: Schema of our network flow database.

Columne	Type	Description
start_ts	8-byte int	start time of flow (seconds elapsed since epoch)
protocol	1-byte int	TCP, UDP, ICMP, etc (mapped to a number)
cli_ip	4-byte int	client IP (converted to int)
srv_ip	4-byte int	server IP (converted to int)
cli_port	2-byte int	client port
srv_port	2-byte int	server port
app	char(30)	application

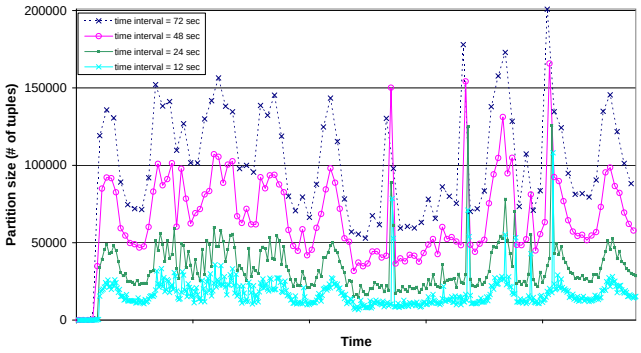

Figure 2: Variation of partition sizes with different partitioning time interval

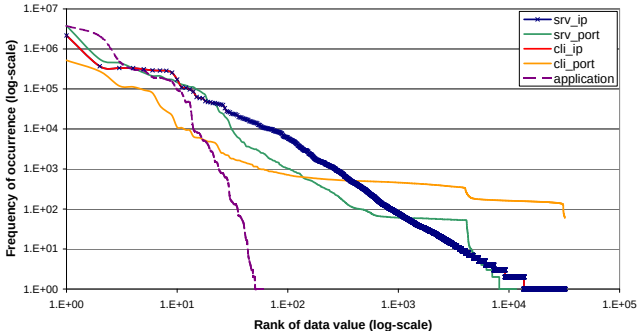

Figure 3: Distribution of data values for different attributes plotted on a log-log scale. They roughly follow the Zipf distribution.

The partitioning procedure could be performed over the time line, the number of tuples and different attributes. To determine along which dimension and how partitioning should be performed, we measured the variation of partition sizes for different partitioning time interval and collected data statistics over informative attributes (*srv_ip*, *srv_port*, *cli_ip*, *cli_port*, *app*).

Figure 2 showed the variation of partition sizes with partitioning time interval of 12, 24, 48 and 72 seconds. It showed large variation in partition sizes over time. In particular, much more traffic was generated in the day time than at night. The statistics also gave us some hints for deciding on the partition boundaries. We could also see that a smaller partitioning time interval was more favorable as it resulted in less variation in partition sizes, ensuring more uniform performance in our system. However, small partitions would incur more overhead in partition management and impact performance. Therefore, we decided to experiment with different partitioning time interval in our project.

Figure 3 shows the data statistics over the *srv_ip*, *srv_port*, *cli_ip*, *cli_port* and *app* attribute. The graph was plotted with a log-log scale. We observed that the distribution of data values of the *srv_ip*, *srv_port*, *cli_ip* and *cli_port* attributes roughly followed a Zipf curve. The distribution of the *app* attribute was less skewed, but it also had a few dominating data values and many unpopular ones. These properties made partitioning over attributes non-uniform and unsuitable. The statistics also gave us insights in indexing. We could see the cardinality of the attributes were different. It would be interesting to evaluate the effect of cardinality on the indexing time.

4.3 Data insertion

To cope with the high data insertion rate of a NIDS, we needed a mechanism that could insert data in batches at high speed. Microsoft SQL Server provided us with a bulk insert command which could import a data file into a database table in a user-specified format. We experimented with it using the data trace 1 that we described above. A number of optimization techniques were used to maximize the data insertion rate. Firstly, we altered the database recovery model from "full" to "simple". This minimized the amount of data that had to be written to the transaction log. For example, it would only record "A bulk insert command was issued" instead of the exact position and data involved. Although this would risk losing all the data upon system crash, it would not matter for our application since one copy of the data was kept in the data trace files. Secondly, a table lock was used instead of individual row locks. This reduced the overhead of acquiring and releasing locks tremendously. Thirdly, we varied the batch sizes to tradeoff the overhead of setting up the batches and the granularity of interleaving the operations. Fourthly, we compressed the original network flow data by some encoding tricks. For example, we transformed the *ip* and *mac_address* attributes from string to integers (actually we mapped all the attributes into integers, except for the *application* attribute). All these techniques were very effective and could increase the throughput by a hundred fold.

4.4 Indexing

In an NIDS system, forensic queries usually involve predicates over multiple attributes. In particular, *start_ts*, *srv_ip*, *srv_port*, *cli_ip*, *cli_port*, and *app* appear in most forensic queries. Thus indexing over these attributes will significantly improve the query performance. Yet the indexing overhead and the performance gain in query processing will depend on the way the indexes are built.

In general, SQL Server supports two kinds of indexes: clustered and unclustered. Clustered index will sort the original table according to the indexing key and store them in this order physically, thus it can provide efficient index scan and support range queries well. For unclustered index, the most common kind is hash index which supports equality predicates well. However, unclustered index do not sort the table physically. Instead it maintains pointers to the data and a scan will involve a traversal over these pointers. Because of the property of these indexes, there can only be one clustered index, but more than one unclustered index on a table. If the database engine can make use of a clustered index to process a range query, it can perform a sequential scan on the disk to retrieve the data and does not need to traverse over many data pointers as with unclustered index. This greatly speeds up the query processing time because memory and disk are slow.

Although clustered index performs much better than unclustered one, building clustered index is usually more expensive. In our system, we chose to build a clustered index for *start_ts* since it appears in most range predicates and unclustered indexes for the other main attributes shown above since they are typically used in equality predicates.

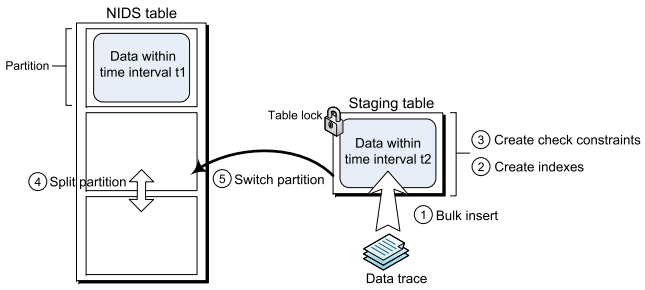

Figure 4: The data partitioning procedure

4.5 Partitioning

Data partitioning involved breaking a table into smaller ones so that data insertion, indexing and query processing could be decoupled and the number of tuples needed to be scanned for query processing could be minimized. The procedures for creating a partitioned table in Microsoft SQL involved creating a partition function, a partition scheme with that function and finally a table with that scheme.

We decided to use *start_ts*, the starting time of a network flow, as the partitioning key because network forensic queries often involved searching for tuples within a certain time frame where an attack was launched. With this partitioning scheme, the query processing unit only needed to scan the partitions relevant to that query instead of the whole table, improving the response time dramatically. This was called partition elimination. We broke down the table into partitions using fixed time interval. Figure 4 shows the data partitioning procedure.

The NIDS table was a partitioned table where all the network flow data was stored. Initially, it was empty with no partition in it. The staging table was a temporary non-partitioned table used to aid the data insertion process. Both tables had the same schema which was required for moving data between them. The partitioning steps were: (1) Data from a network trace would be bulk inserted into the staging table in one batch by acquiring a table lock instead of row locks. Each batch would contain data within a fixed time interval, say with *start_ts* in between T_i and T_j. (2) Indexes could be built on top of the data in the staging table. This operation would be faster than creating the indexes during data insertion, where the latter case involved reconstructing the indexes every time a new tuple was inserted. We had chosen to build a clustered index on *start_ts* and unclustered indexes for 5 other main attributes in the schema. (3) Check constraints would be added to the staging table, specifying the time interval to which the data belonged. In this case, the constraint would be $T_i <$ *start_ts* $<= T_j$. (4) A new partition would be created in the NIDS table by splitting the range of the latest partition into two. This resulting range must match that of the check constraints specified in the staging table. In this example, the initial range was -∞ to ∞. After splitting at T_j, two partitions with ranges (-∞, T_j] and (T_j, ∞] respectively would be created. The data in the staging table could be moved to the (-∞, T_j] partition in the next step as the constraint $T_i <$ *start_ts* $<= T_j$ matched it. (5) By calling a switch partition command, all the data in the staging table would be moved to the new partition (-∞, T_j]. Since only meta-data in the database was involved in this step, the operation was very fast. The next batch of data could then be bulk inserted into the staging table and the above steps (1)-(5) would be repeated.

This insert and switch procedure was used instead of directly bulk inserting all the data into the partitions of the NIDS table because of the following reasons. 1) If we wanted to do bulk insert and query processing in the NIDS table at the same time, row locks instead of table lock had to be used to provide fine-grained data access. This would be inefficient given that we were dealing with a huge number of tuples. 2) The drawback was related to the creation of indexes that was mentioned above. There was no command for us to specify the creation of indexes on a particular partition. It could only be created on the whole table, which meant that we had to create the indexes during data insertion and modify them on each insertion. Since both of these restrictions would slow down the data insertion rate, we had chosen to use the insert and switch procedure instead.

5. EXPERIMENTS

In this work, we designed experiments to evaluate our system on the performance of three important tasks: indexing, data insertion, and query processing. Indexing is crucial to query performance, but its overhead has a significant impact on the insertion rate. The two main goals for our overall system performance are high data insertion rate and fast query processing time. We will describe the evaluation methods and experimental results in details in the following sections.

5.1 Evaluation for indexing

To evaluate the overhead of index construction, we designed the following indexing experiments:
1) Indexing over different data types: int vs. string.
2) Indexing over attributes in same data type with different cardinality.
3) Indexing over different number of attributes.
4) Indexing over different size of partitions.

We have built clustered indexes on the attribtues *start_ts* (8-byte int), *srv_ip* (4-byte int), *srv_port* (2-byte int), *cli_ip* (4-byte int), *cli_port* (2-byte int) and *app* (string) and a composite index over all these 6 attributes in the order (*start_ts*, *srv_ip*, *cli_ip*, *srv_port*, *cli_port*, *app*). We assumed a data insertion rate of 3000 tuple/sec and evaluated the indexing performance on partition sizes of 15 minutes, 30 minutes and 60 minutes. We also assumed that the data partitions have been filled up before we built the indexes, and they remained unchanged during indexing. The results are shown in Figure 5 .

We can learn a few things from this experiment. First, we can see that the indexing time for string data type is longer than that for int data type. This is reasonable since clustered indexing needs to sort the whole table according to the indexing key and sorting string attributes is certainly more costly than sorting int attributes. Thus we may avoid indexing the application attribute for performance reason. Second, even though the distribution of *start_ts*, *srv_ip*, *srv_port*, *cli_ip*, *cli_port* are quite different, they require similar amount of indexing time. This is because we were building indexes over static partitions. And in fact, data distribution can greatly affect the insertion rate if an index already exists on a table during insertion. Lastly, it is interesting to see that the indexing time for the composite index of all at-

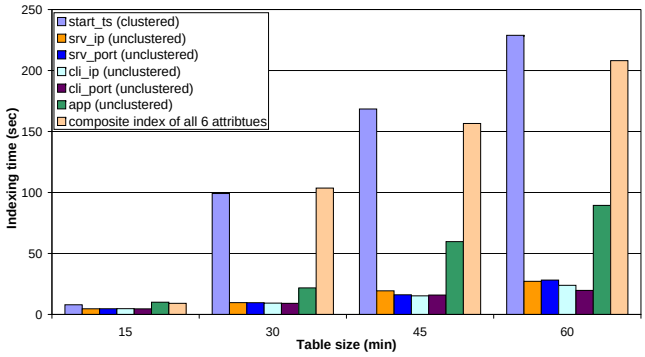

Figure 5: Indexing creation time for different attributes and different table sizes Building a clustered index takes much more time than a unclustered one

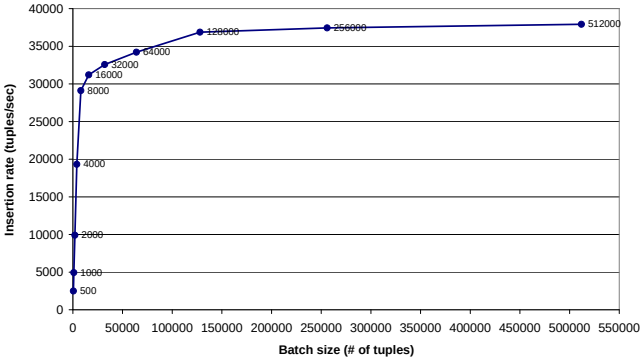

Figure 6: The insertion rate with different batch sizes

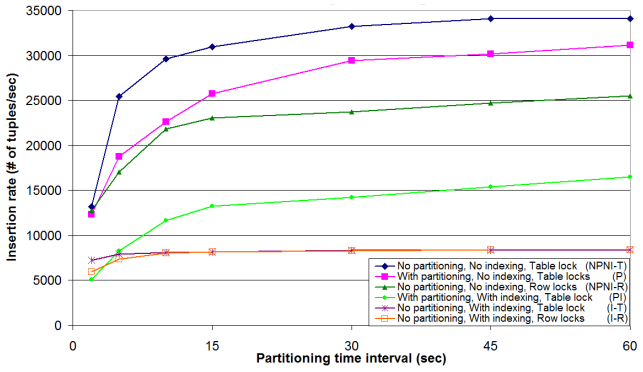

Figure 7: The insertion rate for the overall system with different partition sizes and different schemes

tributes is less than that for the application attribute alone. As sorting is the dominating factor in creating clustered index, we predict this is related to the way the composite index sorts the rows. When the composite index is being created, the rows will be first sorted on the first attribute start_ts and then on the second attribute if the rows have the same start_ts value and so on for the other attributes. Since most rows have distinct start_ts value in this trace, after sorting on start_ts, few additional sorting on other attributes needs to be performed. Sorting on the application attribute is unlikely to happen since it is the last attribute on the composite index. Thus, creating the composite index requires fewer costly string comparisons and could be built faster.

5.2 Evaluation for insertion

We first run a micro-benchmark to study how the insertion rate is affected by the batch size and then investigate more deeply on the insertion rate of different schemes implemented in our system.

Figure 6 shows the insertion rate with different batch size settings. As seen from the graph, the insertion rate increases non-linearly with increasing batch sizes. This is because the bulk insert command is processed by treating each batch as one transaction. The larger the batch is, the fewer transactions it will be required to perform, thus minimizing the overhead and achieving a higher throughput. A batch size of at least 50000 tuples is needed to achieve a high insertion rate.

Figure 7 shows the data insertion rate of different schemes with different partition sizes. In the graph, the insertion rates for all schemes increase non-linearly with increasing partition sizes because fewer transactions need to be performed for larger partition sizes. This agrees with the results in Figure 6. Among the different schemes, NPNI-T shows the highest insertion rate, with a maximum of nearly 35000 tuples/sec. We can consider this as the ideal insertion rate that partitioning and indexing schemes can achieve if there is no overhead. However, NPNI-T is unrealistic for NIDS applications because by acquiring a table lock, query processing cannot be performed at the same time. If row locks are acquired instead, as in NPNI-R, the insertion rate will be dropped by 10000 tuples/sec due to more locking overhead. Next, if partitioning is used (in the P scheme), the insertion rate will be 5000 tuples/sec slower than that of NPNI-T because of partition splitting and switching. However, this overhead is not very significant as we see that the insertion rate of P is still faster than that of NPNI-R. We have only evaluated P with table lock, but not with row locks, since partitioning enables the use of table lock by decoupling insertion and query processing. It is unreasonable to use row locks, which have worse performance, if table lock can be used. If both partitioning and indexing are used (PI), the insertion rate drops down to 15000 tuples/sec because we are building 6 indexes which require much time. Despite the slow down, the performance gain in query processing is expected to outweigh the overhead of index creation. Lastly, the slowest schemes are with indexing only (I-T and I-R) with table lock and row locks respectively. It is because with indexes over the whole table without partitioning, all these indexes have to be rebuilt after each bulk insertion. The overhead is so large that using table lock or row locks do not affect the result.

From these experiments, we observe that RDBMS system can achieve the high data insertion rate required for NIDS application. Even with the partitioning and indexing overhead, the high insertion rate can still be sustained.

5.3 Evaluation for query processing

To show the benefit of partitioning and indexing in query processing, we have executed 6 queries in our system with and without data being inserted at the same time. Q1-Q4 are simple queries that allow us to evaluate different schemes easily. Q1 selects all the traffic within a certain time range.

It allows a network administrator to check all the recent traffic in details. Q2 selects the number of flows sent from a host with IP x. This can be used to inspect any suspicious hosts. Q3 selects all the traffic running an instance of application a, enabling an administrator to narrow down suspicious hosts in case he knows that there is a vulnerability in application a. Q4, which is a combination of Q1 and Q2, finds the number of flows sent from a host on a certain port within a certain time range.

Q1 - on time range:
SELECT * FROM NIDS
WHERE start_ts >= t;

Q2 - on IP:
SELECT count(*) FROM NIDS
WHERE srv_ip = x;

Q3 - on application:
SELECT * FROM NIDS
WHERE app = a;

Q4 - on time, IP and port:
SELECT count(*) FROM NIDS
WHERE start_ts >= t AND srv_ip = x AND srv_port = p;

We have also tested our system further by issuing more complicated real-life network forensic queries. Q5 is an example that looks for backdoor intrusions. It finds 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. Q6 is an important query used in network forensic analysis. It searches for all the hosts infected by a malicious host. It traces the scope of the infection using a signature of an attack, sending out traffic through a certain port. In real NIDS, this is in the form of a recursive query because the infected hosts will in turn become malicious and infected other hosts. We will need to trace the connection tree exhaustively. However, due to time limitation of this project, our query only searches for infected hosts directly connected to the malicious source.

Q5 - on backdoor intrusion:
SELECT * FROM NIDS a, NIDS b
WHERE a.srv_ip = b.cli_ip
  AND a.cli_ip = b.srv_ip
  AND a.srv_port = p
  AND a.start_ts > t1
  AND a.start_ts < b.start_ts
  AND b.start_ts - a.start_ts < t2;

Q6 - searching for infected hosts:
SELECT start_ts, srv_ip,srv_port, cli_ip, cli_port
FROM NIDS,
  (SELECT start_ts, srv_ip,srv_port, cli_ip, cli_port
  FROM NIDS
  WHERE srv_ip = x
    AND start_ts >= t1
    AND srv_port = p) as INFECTED
WHERE NIDS.srv_ip = INFECTED.cli_ip
  AND NIDS.start_ts > INFECTED.start_ts
  AND NIDS.start_ts - INFECTED.start_ts < t2
  AND NIDS.srv_port = p;

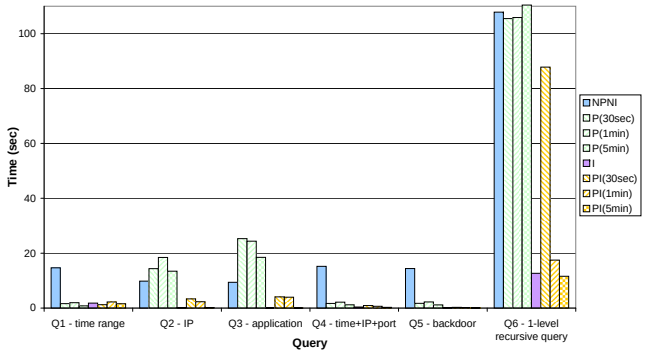

Figure 8: Query performance (time) without data insertion

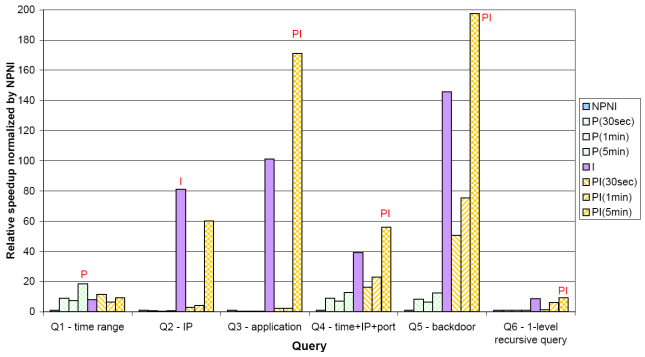

Figure 9: Query performance (speedup) without data insertion

Figure 8 and 9 shows the query processing performance of the four partitioning and indexing schemes: NPNI, P, I and PI. For the partitioning schemes, several partitioning time interval are also evaluated, abbreviated as P(time interval). In this experiment we do not insert any data into the table during query processing. We first construct tables with different schemes, filled it with 45 mins of network flow data from trace 2 and then run each of these queries over it. We call this the offline mode. In this way, by removing the partitioning and indexing overhead, it can measure the net performance gain brought by partitioning and indexing techniques. Figure 8 shows the actual query processing time used to execute each query while Figure 9 shows the relative performance speedup with reference to the NPNI scheme. The scheme with the best performance for each query is labeled in Figure 9. It can easily be seen that PI, P and I outperform NPNI by a factor of 10 to 200. For Q1, P(5 min) has the lowest query processing time because it is a query on time range only. Accessing the time range using partition is more effective than using the start_ts index. Moreover, the partition size 5 minutes matches with the query data range, giving the best performance. For Q2 and Q3, both I and PI(5min) perform the best because the query involves searching for a predicate and indexing can greatly improve performance. On the other hand, the P schemes actually have a worse performance than NPNI because partitioning on time cannot help in searching for the IP and application predicate and the overhead of partitioning will degrade the

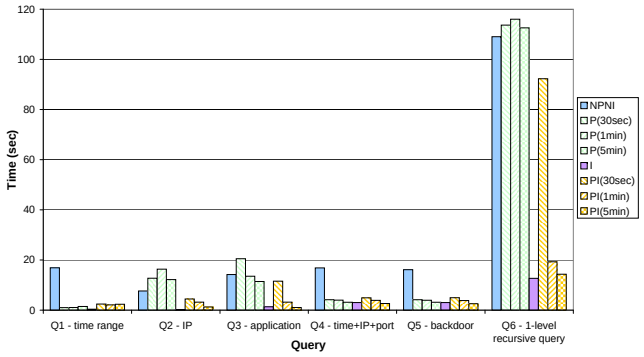

Figure 10: Query performance (time) with data insertion

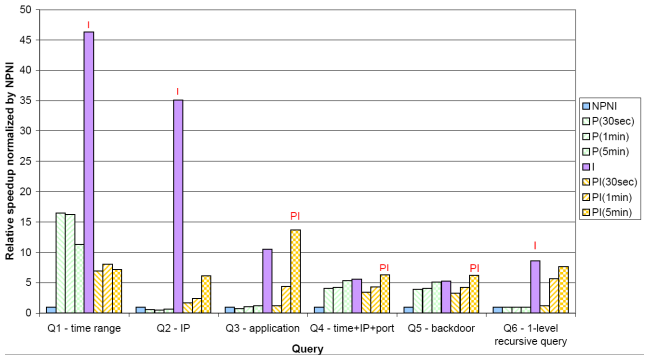

Figure 11: Query performance (speedup) with data insertion

performance. For Q4 to Q6, PI(5min) gives the best result since the queries involves both time ranges and query predicates. We also observe that the partition size has to match well with the query data range for better performance. In this experiment, a larger partition size usually works better because our queries focus on a large time window. In general, partitioning and indexing benefits most types of queries.

To simulate a realistic NIDS, we ran our system under the online mode, where it performed query processing and data insertion at the same time. During the experiment, data was continuously being inserted into the table at a rate of 3000 tuples/sec. After the data size had reached a certain point, we issued the query. Note that data was still being inserted at the same rate while the query was being processed. This would impose substantial load on our system. Figure 10 and 11 shows the query performance of the online mode operation. Again Figure 10 shows the actual processing time whereas Figure 11 shows the speedup relative to the NPNI scheme. The scheme with the best performance for each query is labeled in Figure 11. Similar to the offline mode, we see that P, I and PI still outperform NPNI in most cases, but by a factor of 10-45 times, given the overhead of partitioning and indexing. It is interesting to see that I perform much better for Q1 and Q2. The reason is that I has less overhead than PI and these two queries are too simple. They require so little query processing time that PI cannot bring enough benefit to outweigh its overhead. We expect PI to benefit more for complex queries. This is not a problem since most forensic queries are fairly complicated. One may also argue that if indexing alone can bring substantial speedup, partitioning should not be used. However, as shown in Figure 7, I has the slowest data insertion rate that makes it inappropriate for high data rate network. We have only tested with a data rate of 3000 tuples/sec in this experiment, but we expect I to perform worse if the data rate is increased. For Q3 to Q6, PI(5min) gives the best performance, showing its applicability for optimizing many types of queries.

6. CONCLUSION AND FUTURE WORK

Forensic analysis is crucial in network intrusion detection systems. A historical flow database in an NIDS needs to suport: a) fast data insertion rate; b) low query processing latency; c) online query processing with data insertion.
To meet these challenges, we propose building an NIDS using an "off-the-shelf" relational database system with data partitioning and indexing techniques. Our experimental results have shown that partitioning over proper size with indexing can not only improve the data insertion rate, but also speed up query processing for up to 200 folds.
For future work, it would be interesting to study how the right partition size can be determined adaptively with the incoming query workload, experiment with different data insertion rates and evaluate the query performance for more complex queries. Further speedup for recursive queries could also be achieved by unfolding the recursion and rewriting the query with irrelevant data on the timeline pruned.

7. REFERENCES

[1] Microsoft Corporation Inc. Microsoft Open Database Connectivity (ODBC).
[2] Microsoft Corporation Inc. Microsoft SQL Server 2005.
[3] Microsoft Corporation Inc. Microsoft Windows Server 2003.
[4] Microsoft Corporation Inc. Performance Monitor.
[5] Nathan Folkert et al. Optimizing Refresh of a Set of Materialized Views. In *Industrial Session: Data Warehousing and Data Mining*, pages 1043–1054. Oracle Corporation, 2005.
[6] Stonebraker et. al. C-store: a Column-Oriented DBMS. In *Proceedings of the 2005 VLDB*, pages 553–564, 2005.
[7] R. Geambasu, T. Bragin, J. Jung, and M. Balazinska. On-Demand View Materialization and Indexing for Network Forensic Analysis. In *Proceedings of NetDB 2007*, April 2007.
[8] Sun Microsystems Inc. The Java Database Connectivity (JDBC).
[9] T.J and L.G. Generating a Data Stream Warehouse using Data Depot. In -.