Object-Relational DBMS for Packet-Level Traffic Analysis: Case Study on Performance Optimization

M. SIEKKINEN, E.W. BIERSACK
Institut Eurécom
BP 193
06904 Sophia-Antipolis Cedex
France
Email: {siekkine,erbi}@eurecom.fr

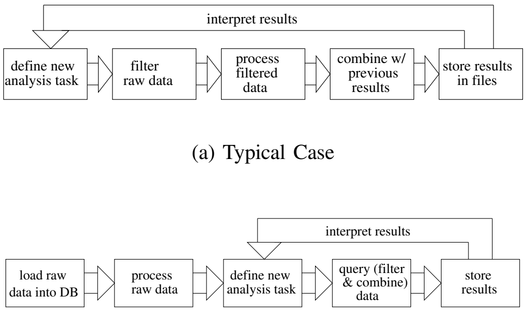
V. GOEBEL
Department of Informatics
University of Oslo
P.O. Box 1080 Blindern
NO-0316 Oslo, Norway
Email: goebel@ifi.uio.no

Abstract— **Analyzing Internet traffic at packet level involves generally large amounts of raw data, derived data, and results from various analysis tasks. In addition, the analysis often proceeds in an iterative manner and is done using ad-hoc methods and many specialized software tools. These facts together lead to severe management problems that we propose to address using a DBMS-based approach, called InTraBase. The challenge that we address in this paper is to have such a database system (DBS) that allows to perform analysis efficiently. Off-the-shelf DBMSs are often considered too heavy and slow for such usage because of their complex transaction management properties that are crucial for the usage that they were originally designed for. We describe in this paper the design choices for a generic DBS for packet-level traffic analysis that enable good performance and describe how we implement them in the case of the InTraBase. Furthermore, we demonstrate their importance through performance measurements on the InTraBase. These results provide valuable insights for researchers who intend to utilize a DBMS for packet-level traffic analysis.**

I. INTRODUCTION

Internet traffic analysis is a discipline that involves typically large amounts of data. In off-line traffic analysis the raw traffic data, such as TCP/IP packet headers, is generally processed and analyzed in many ways. Each analysis task generates new data that needs to be stored and possibly processed again later. In other words, traffic analysis is often an iterative process: A first analysis is performed and based on the results obtained, new analysis goals are defined for the next iteration step. Today, the state of the art of traffic analysis tools is handcrafted scripts and a large number of software tools specialized for a single task. These facts together lead to the following major issues[1]:

Management: We identify two problems: 1) many tasks are solved in an ad-hoc way using scripts that are developed from scratch, instead of developing tools that are easy to reuse and understand and 2) traffic analysis involves large amounts of data. By data we mean not only the traffic traces containing unprocessed packet data, but also all derived data generated by each analysis task. However, the tools used generally do not provide any support for managing these large amounts of data. Therefore, the data is typically archived in plain files in a file system. Depending on the number of files and the skills of the researcher to properly organize them, the later retrieval of a particular trace or data item may be a non-trivial problem. As Paxson [12] has pointed out the researchers themselves often cannot reproduce their own results.



(a) Typical Case

(b) Using DBMS-based Approach

Fig. 1. Cycles of Tasks for the Iterative Process of Off-line Traffic Analysis.

Analysis cycle: A common workflow to analyze network traffic proceeds in cycles (see Figure 1(a)). Since the semantics of the data are not stored during the analysis process, reusing intermediate results becomes cumbersome and usually the process needs to restart again from the raw data after modifying the scripts.

Scalability: Scalability is an important issue in traffic analysis and poses a problem as the amount of data is typically large. Already packet traces larger than 5 GB may pose serious problems for certain tools because of too large memory or run-time requirements. An example is the well-known tool `tcptrace` [3].

In order to cope with the issues detailed above, we have adopted an approach based on an object-relational database management system (DBMS). It was first introduced in [14] where we describe in detail how a DBMS-based method can overcome the above mentioned problems. For example, Figures 1(a) and 1(b) illustrate the advantage of such an approach to the traditional approach through the typical analysis process

[1] These issues are also to some extent discussed in [10] and [9].

TABLE I

CHARACTERISTICS OF DIFFERENT DBMS-BASED APPROACHES FOR TRAFFIC ANALYSIS.

Approach	**Aggregation level**	**Traffic volumes**	**Data mgt**	**Metadata mgt**	**SW mgt**	**Publicly available**	**Integrated approach**	**On-line**
NetLogger/NetMiner [4]	flow	10Gbit/s		X				X
LOBSTER [1]	flow	10Gbit/s		X				X
Gigascope [6] (AT&T)	packet stream	Gbit/s			X			X
Internet Traffic Warehouse [5] (Telcordia)	packet	hundreds of MBytes/day	X	X	X			
IPMon [7] (Sprint Labs)	packet	TBytes		X				
InTraBase (Institut Eurecom)	packet	tens of GBytes/trace	X	X	X	X	X	

X = feature is supported
blank = feature is not at all supported or is implemented in an ad-hoc manner

cycle.

We have gathered significant experience on packet-level TCP traffic analysis with our database system (DBS), called InTraBase, by performing in-depth TCP traffic studies with it [16] [15]. While DBMS-based traffic analysis methods are very attractive because they can overcome the problems with the traditional approaches stated above, *the main challenge is performance*. Such an approach becomes useless if, in practice, the performance does not allow to take full advantage of the features provided by the DBMS to support complex analysis. Because of their complex transaction management properties, DBMSs are often considered too heavy, and, thus, too slow, for intensive packet-level traffic analysis.

A typical off-the-shelf DBMS is optimized for a usage pattern that is very different from the one we have with the InTraBase. For example, the InTraBase generally does not need to process many concurrent queries, and thus, we can relax the parameters affecting the concurrent query processing as much as possible in order to reduce performance overhead. Therefore, tuning the DBMS to fit this specific usage is very important. Furthermore, the characteristics of the traffic data often lead to a certain specific query being extremely popular: "give me all the packets from this connection in chronological order". Hence, to have acceptable performance, the design of the DBS must take the nature of the data into account and focus on optimizing the performance of this popular query.

Table I summarizes the differences between the various existing DBMS-based approaches for traffic analysis. The main characteristics that differentiate InTraBase from the others are: (i) InTraBase does pure *off-line* analysis and does not address the packet capturing or on-line monitoring related issues at all, (ii) InTraBase is designed for intensive *packet-level* analysis on (iii) *moderate size* ($<$ 50 GB) traffic traces. It is not a tool to do, for instance, real-time network health monitoring for a large ISP due to the immense amounts of data that would need to be treated constantly, but it is rather a research tool for fine-grained analysis of Internet traffic. We refer the reader to [14] for more details on the comparison to the other approaches listed in Table I.

We describe in this paper the main principle when designing a DBS to support efficient analysis of packet-level network traffic measurements and how to enforce this principle. We base the design decisions on the characteristics of this measurement data and the typical analysis process cycle. We demonstrate the importance of correct design through performance measurements in the particular case of managing TCP/IP packet traces with the InTraBase.

In Sections II and III we describe first in a general manner the important issues to consider when managing packet-level traffic data with a DBMS, and then explain for each particular issue how it is taken into account in the case of the InTraBase. We discuss results from performance measurements in Section IV and finally conclude with Section V.

II. MANAGING PACKET-LEVEL TRAFFIC DATA WITH A DBMS

A. Storing and Querying Packets

Packet-level traffic data is commonly recorded in plain files as packet *traces* which may contain hundreds of millions of packets and millions of connections each. When storing this data into the database, it is out of the question to store packets from each connection into a separate table given the potential large number of connections in a typical packet trace file. The reason is that handling millions of tables in a single database becomes very inefficient. In addition, querying more than a single connection at once becomes very cumbersome since each additional connection means joining an additional table into the query. Storing packets from each trace into the same table would eventually lead to performance problems when the table size grows excessively. Hence, a logical approach is to store packets from each trace into a separate table. In this way, since the objective is to analyze packet traces smaller than 50 Gbytes, the maximum table size stays reasonable.

The analysis tasks are generally performed for predefined groups of packets within a trace. In the case of TCP traffic, this group is typically a connection or a flow[2]. The latter can be used also for traffic generated by a connectionless protocol such as UDP. Having done extensively TCP traffic analysis, the most common query that we have used is:

```
SELECT *  FROM trace1_packets
          WHERE connection_id=x
          ORDER BY timestamp
```

The above query, hereafter referred to as the c-query (from connection and common), returns all the attributes (e.g. for TCP traffic all the IP and TCP header fields and a timestamp) of all the packets that belong to connection x from table

[2] A flow is usually defined as a part of a connection using a timeout or a maximum packet count.

trace1_packets in chronological order. As indicated in Figure 2, this query is executed in the beginning of each typical analysis task and the analysis results, e.g. the number of reordered packets, are computed by inspecting the query results row by row. Finally, the result is stored into another table or alternatively printed on the screen. Note that even if the c-query is made more complex (e.g. by adding WHERE rules or by joining in another table), the query processor of the DBMS would translate it into the original c-query and perform additional filtering and querying separately and merge the results in the end. In other words, fetching packets for a specified connection from the database boils every time down to executing the c-query. We ignore the common queries that do not involve querying packet-level data because their contribution to the performance is negligible due to orders of magnitudes smaller data sets. For example, before executing analysis tasks in the way described in Figure 2, one may wish to select a set of connections having certain characteristics (e.g. more than 100 packets) for which these analysis tasks are performed. The table storing per-connection statistics contains only a very small fraction of the amount of rows that are stored in the table holding the packets.

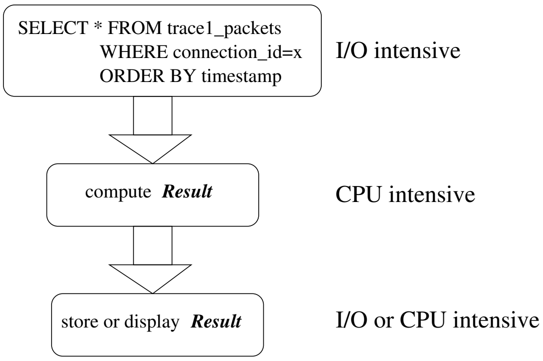


Fig. 2. Executing a typical analysis task.

Case Study on InTraBase:
InTraBase runs on PostgreSQL DBMS mainly because of its object-relational nature that allows to extend the functionality through procedural language (PL) functions [14]. The layout of the tables of the InTraBase are shown in Figure 3. We can divide the tables into *base tables* and *RCA (Root Cause Analysis)*[3] *tables.*

Base tables are packets, connections, traces, and cid2tuple. The table traces contains annotations about all the packet traces that are uploaded in the database. The packets table holds all packets for a single trace. The table connections holds connection level summary data for all traces. The cnxid attribute identifies a single connection in a packet trace, reverse differentiates between the two directions of traffic within a connection, and tid identifies a single trace. Cid2tuple is a table to store a mapping between unique cnxids and 4-tuples formed by source and destination IP addresses and TCP ports. The attributes of the packets table are directly from the verbose output of tcpdump for TCP packets. The attributes of the other tables were chosen.

The remaining tables form the set of RCA tables. Each connection may be partitioned into several *bulk transfer periods* (BTP) and *application limited periods* (ALP) which are stored into the bulk_transfer and app_period tables, respectively. In addition, the n_lim parameter can have values from 0 to 1 with 0.05 steps for each connection. For details about the BTPs and ALPs and the n_lim parameter, we refer the reader to [15]. Each BTP stored in the bulk_transfer table has additional characteristics computed and stored into the bnbw_test, rwnd_test, and retr_test tables.

Populating the base tables has been carefully explained in [14]. The RCA tables are populated with the help of procedural language (PL) functions written in PL/pgSQL and PL/r, and C-language functions. Defining PL and C-language functions is one of the ways in which PostgreSQL allows to extend the functionality of the object-relational DBS. Each function operates in the way described in Figure 2. It queries packets for a given connection, computes one or several results, and stores them into one of the RCA tables.

B. Tuning the DBMS

A standard off-the-shelf DBMS is optimized for processing a large number of concurrent queries. It takes care of issues related to parallel access to data and enforces atomicity of transactions, etc. However, our experiences with the InTraBase suggest that the typical usage of a packet-level traffic analysis DBS for research purposes generates a very different workload: few users seldom issuing queries that commonly are very I/O intensive touching large amounts of data (see Section II-A for the c-query). In addition, several queries are rarely executed simultaneously. In this case the DBMS must be tuned to conform to the special usage.

The fact that concurrent query processing is rare allows to set most of the buffer sizes high since they are normally defined on per process, i.e. query, basis. For the majority of DBMSs, it is possible to tune parameters such as memory available for a sorting or index creation. Since the c-query involves an ORDER BY operation and the queried connection can contain up to millions of packets, it is important to set the amount of memory available for sorting as high as possible.

Write-Ahead Logging (WAL), a.k.a. redo logging [8], is a standard approach to transaction logging in DBMSs. Since we can assume little parallel access to data, the WAL parameters can be set as lazy as possible, by setting the commit delays to the maximum, in order to let the underlying operating system (OS) optimize the I/O operations.

Caching plays a very important role in the performance of the DBS. The amount of memory available for caching is also a modifiable parameter in most of the DBMSs. Caching can greatly improve the performance of per-connection or per-flow querying of packets in cases where same groups of packets,

[3] The naming comes from the fact that they are used for TCP Root Cause Analysis [16].

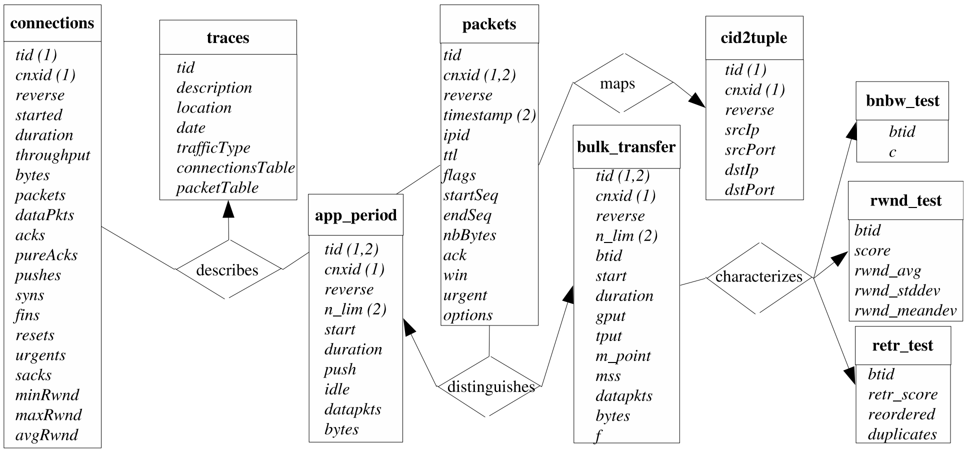


Fig. 3. The Layout of the Core Tables in the InTraBase. Indexes are denoted with numbers in parenthesis following the attribute: e.g. table `connections` has an index consisting of `tid` and `cnxid` attributes.

i.e. specific connections, are analyzed over and over again in different analysis tasks. In this case the data would be read from the disk once and cached for the following queries. Clearly, the execution order of the queries needs to be carefully chosen. In addition, as we explain in the next section, the caching techniques used need also to be well chosen.

Case Study on InTraBase:
The InTraBase is running on PostgreSQL on top of Linux 2.6.3. The hardware consists of an Intel Xeon Biprocessor 2.2 GHz with a SCSI RAID system and 6 GB RAM. PostgreSQL allows to set the parameter `work_mem` that controls the amount of memory available for internal sort operations and hash tables. We set this to no higher than 1.5 GB in order to avoid out-of-memory problems: On a 32-bit system the maximum process size is around 3 GB and in certain cases several sort operations (typically not more than two) may be run in parallel each of which is allowed to consume the amount of memory specified by `work_mem`. Also, 1.5 GB should be sufficient in most cases to sort in main memory all the packets of a single connection. WAL commit delays were set to 100 ms (the maximum).

III. Minimizing the cost of I/O operations

Our goal is to optimize the execution of an analysis task displayed in Figure 2. Optimizing the middle step in Figure 2 is always specific to the analysis task and, therefore, generic solutions for that do not exist. Based on our experience, regardless of the analysis task, the number of results stored or displayed per task is typically very small compared to the number of packets queried in the first step. Thus, the last step in Figure 2 is rarely the bottleneck and it is the first step that we focus on. This step reads the tuples that represent the packets from disk and it is therefore generally I/O bound. We derive from the above the main design principle:

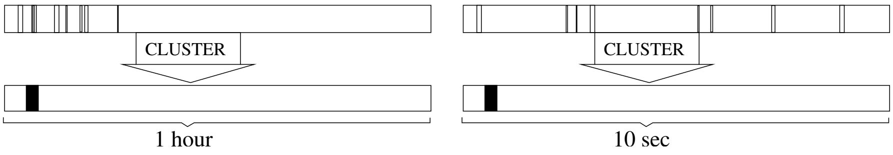
- *Minimize the I/O time for the c-query.*

A. Indexes for Fast Lookup

The two most important concepts in DBMSs for I/O optimization are *indexing* and *clustering* [13]. Indexes allow fast lookup of specific rows from tables. They can be thought of as hash lookups of logical records. In the case of our c-query, since each queried packet belongs to the same group, we can increase the performance by adding an index on the connection or flow identifier to each table containing packets. This enables the DBMS to do an *index scan* with the help of the created index, touching only the required disk blocks, instead of doing a *sequential scan* on the contents of the entire table, touching all the disk blocks associated with the table, each time the query is executed. Since the c-query includes ordering by timestamp, one might think that it would be beneficial to create another index on the timestamp attribute or add the timestamp into the index on connection identifier. In this way, the DBMS can perform simply an index scan on the new index and does not need to sort in addition. There is a caveat, though: Index scanning using this combined index is computationally much more expensive than with the simple connection identifier index. With InTraBase, the query processor of the DBMS reports almost hundredfold per-row cost estimates for the index scan using the combined index when compared to the plain connection identifier index. Therefore, the combined index is useful only when a small fraction of packets of the connection is queried. We have found it useful when studying the option negotiation (e.g. MSS and window scaling) during the TCP connection establishment, for instance.

B. Clustering to Minimize Cost of I/O Reads

Clustering means grouping together data by physically reordering it on the hard disk. The advantage is improved performance for operations that access the grouped data due to accessing only adjacent disk blocks instead of scanning through the entire disk in the worst case. The speedup may



(a) Trace captured at a slow link.

(b) Trace captured at a fast link.

Fig. 4. The effect of clustering with two different types of traffic traces of the same size. Black stripes are packets belonging to a given connection and their horizontal distance from each other reflects the physical distance on the disk.

sometimes be tremendous but depends highly on the characteristics of the data and the clustering parameters. For the case of packet-level traffic data, consider Figure 4 that illustrates the effect of clustering through two different examples. The two traffic traces are of the same size in terms of number of packets but have different durations. In order to speed up the c-query, we cluster the packet data with respect to connections or flows. The packets of each traffic trace are originally stored on the disk in their arrival order. That is why the impact of clustering on a packet trace captured on a high speed link with numerous parallel connections, Figure 4(b), is much bigger than on a trace captured at a low speed link with only few parallel connections, Figure 4(a). It is clear that the penalty of the random seeks necessary for reading all the packets of the connection in the unclustered case compared to the sequential reads in the clustered case is much higher in Figure 4(b) than in Figure 4(a).

C. Parallel I/O

The I/O time can further be reduced by means of parallel I/O. There are several ways to go. It is possible to implement it on application level with parallel DBSs. This approach gives a lot of control over the parallelism through load balancing, for instance, but may suffer from severe overhead due to complexity (e.g. distributed transaction handling). A simpler approach is to implement it on the lowest layer possible, that is, using RAID (redundant array of independent disks) on striping mode. RAID striping is typically implemented so that adjacent blocks are written on different disks in order to maximize the parallelism in the case of sequential disk access. For example, when striping over n disks, any n adjacent disk blocks would be stored on different disks. Since we have already clustered the data in connections, and thus, access adjacent disk blocks whenever executing the c-query, we obtain maximal parallelism with I/O operations.

D. Caching

In addition to minimizing the I/O time of the c-query, it is equally important to avoid repeating it unnecessarily. Therefore, caching is important. If the same c-query is executed several times in a row, the query results should remain cached in the main memory after the first query in order to avoid repeating the expensive I/O operations. It is possible to cache the data on different layers: on the OS layer or on the application layer. All modern OSs implement some sort of caching techniques. The same applies to most modern DBMSs. However, they may all use different techniques and, therefore, implementing this principle is specific to each DBS. We will describe the case for the InTraBase in the next section.

E. Case Study on InTraBase

We have addressed in [14] the performance of populating the base tables as well as the disk space consumption when storing packet-level data in a database. Populating the RCA tables boils down to executing several times analysis tasks described in Figure 2. Similarly does most of subsequent "manual" analysis tasks that we do, such as plotting different parameters (throughput, sent and received packets etc.) of a specific connection against time in order to understand its evolution.

To comply with the main principle, we have indexed and clustered some of the tables on the InTraBase. The indexes of the tables are marked with numbers in parenthesis following the attribute. The `packets` table is clustered based on the index on `cnxid`. We would not gain much by clustering the data in the other tables since in the other tables each index value returns only a few rows, i.e. two rows per connection or a maximum of a few hundred rows with `app_period` and `bulk_transfer` tables in the case of a very large connection. Also, these tables would need a periodical reclustering since the contents are changing when ever a new packet trace is inserted into the system. The server used for the InTraBase is operating a RAID consisting of eight SCSI disks capable of running in striping mode.

Indexing and clustering cause some additional overhead: indexes consume disk space and clustering is a rather expensive operation. We observed in [14] that the indexes cause approximately 15% overhead in disk space consumption. Clustering a 5 GB packet trace took us 28 minutes and creating an index for the `cnxid` attribute for the clustered table 5 minutes with the InTraBase[4]. Both are acceptable considering that they are *one time* operations and that the potential gain is enormous as we will demonstrate in Section IV.

[4] PostgreSQL provides a `CLUSTER` operator that is very slow. A faster way to cluster a table with PostgreSQL is to create a new one from query results, i.e. by executing `CREATE TABLE newtable AS SELECT * FROM oldtable ORDER BY cnxid` and then recreate the indexes for the `newtable` table (see the manual [2]).

What comes to caching, it is only useful if the same data is repeatedly queried. Therefore, in the case of InTraBase, it is particularly important when populating the RCA tables where each connection is analyzed several times with independent analysis tasks. Naturally, the order of execution needs to be such that all analysis tasks are executed for a single connection sequentially. As for where to cache, we have several ways to go with the InTraBase that is running on PostgreSQL on top of Linux. PostgreSQL has two separate caching methods: it uses its own buffer cache, implemented as ARC (Adaptive Replacement Cache [11]) starting from version 8.0, but also relies on the file system cache of the underlying OS. ARC tries to avoid cache flushing (caused by a one-time very large sequential scan, for instance) by keeping track of how frequently and how recently pages have been used and keeps in the cache pages that have the "best" balance of the two. Since we would like to cache every time the query results of the c-query, cache flushing would be in fact desireable each time we execute a new c-query. Thus, we minimized the utilization of PostgreSQL's own cache and attempt to force the DBMS to use Linux file system caching as much as possible.

IV. Performance Measurements

In order to demonstrate the importance of the I/O optimizations described in Section III, we measured several metrics in the case of the InTraBase. To quantify the impact of individual optimizations on the performance, we did measurements while executing the c-query with and without indexing, clustering, and RAID striping. The query was executed on thousands of connections with different sizes in number of packets. The query results were directed into /dev/null in order to measure the retrieval process of the packets only. We measured also the effect of caching when executing several times the same most common query.

We used two different packet traces: one recorded on a Gigabit link on a university edge containing a mixture of Internet traffic, and another one recorded on a much slower link with a total throughput all the time below 10 Mbit/s containing only BitTorrent traffic. In this way we could observe the difference in clustering visualized in Figure 4. We selected from the Gigabit trace the 3060 connections having more than 100 packets and a different number of packets. From the BitTorrent trace we selected all the 583 connections having at least 100 packets. In the Gigabit trace, we had only few connections with very large numbers ($> 10^6$) of packets.

A. Impact of Indexing and Clustering

Table II contains the means of the measured values: exec time is the total execution time measured in wall clock time, CPU iowait is the time that the CPU was idle during which the system had an outstanding disk I/O request, CPU system is the CPU utilization time spent executing tasks at the kernel level, and number of sectors are those read from the hard disk (the size of a sector is 512 bytes). If the packet table is not indexed by the connections, the DBMS is forced to read through all the packets of the table when executing the most common query. Thus, the execution time should be more or less constant regardless of the number of packets queried[5]. When we indexed the data by connections, the means dropped dramatically. Clustering the indexed data had again a similar effect.

TABLE II
Average Values of the Measurements.

Gigabit Trace				
test case	**exec time (s)**	**CPU iowait (s)**	**CPU system (s)**	**sectors read**
1	222	160	20.4	-
2	6.07	5.22	0.338	30600
3	0.524	0.050	0.031	2160

BitTorrent Trace				
test case	**exec time (s)**	**CPU iowait (s)**	**CPU system (s)**	**sectors read**
1	223	168	19.1	7280000
2	7.81	3.96	0.529	42200
3	3.71	0.486	0.255	20400

test cases: 1) unindexed, unclustered, 2) indexed, unclustered, 3) indexed, clustered

Figures 5 and 6 reveal that while it is always good to use an index when querying a small number of packets, it is not necessarily the case when querying a large number of packets unless the data is clustered. In the case of the Gigabit trace, an index with unclustered data becomes virtually useless after the number of queried packets reaches a few hundreds of thousands. In the case of the BitTorrent trace, using an index proves to be always beneficial. The difference with respect to the monitored link speed between the two types of the traces used, explained in Section III (Figure 4), is clearly visible in Figures 5 and 6: clustering has a much bigger impact on the Gigabit trace than on the BitTorrent trace captured on a low-speed link. In the plots for the Gigabit trace, we expect similar variation in the measured values with the larger connections as well, there simply were not many samples of those connections for it to be visible in these plots. In the case that data is clustered according to the index, the total execution time of the query scales more or less linearly, as expected. These observations are even more clearly visible in Figures 7 and 8 that show the CPU's iowait part of the total execution time. The Figures 9 and 10 demonstrate that, in addition to the fact that the read head needs to move a lot more when seeking for unclustered data, many more sectors are generally required to be read from the disk.

Without indexing and clustering the time to execute the c-query of a single analysis task for each analyzed connection of the Gigabit trace would take approximately eight days. As for the BitTorrent trace, it would take one and a half days. Without I/O optimizations the total execution time is linearly dependent on the number of connections. When introducing an index the total execution times drop to 5 h and 1.3 h, respectively. Finally, when the data is additionally clustered we obtain total execution times of 27 min and 36 min, respectively. We observe that after indexing and clustering the execution time is

[5] In the case of the Gigabit trace, we executed on a few randomly chosen connections since it would have taken too long with all the 3060 connections without indexing.

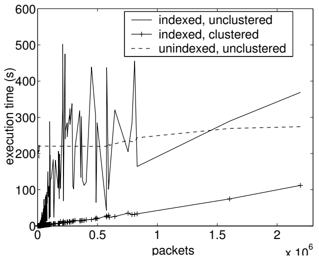


Fig. 5. Total execution time of the c-query for the Gigabit trace.

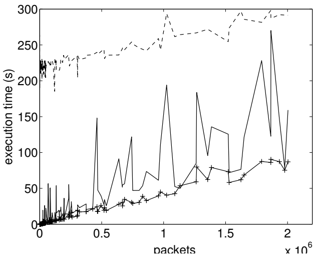


Fig. 6. Total execution time of the c-query for the BitTorrent trace.

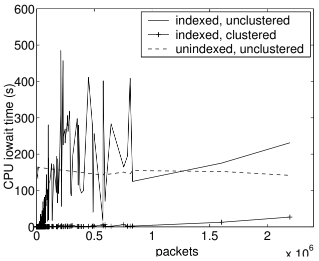


Fig. 7. CPU iowait time of the c-query for the Gigabit trace.

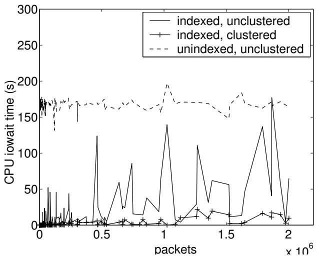


Fig. 8. CPU iowait time of the c-query for the BitTorrent trace.

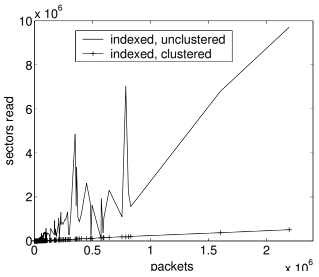


Fig. 9. Number of sectors read when executing the c-query for the Gigabit trace.

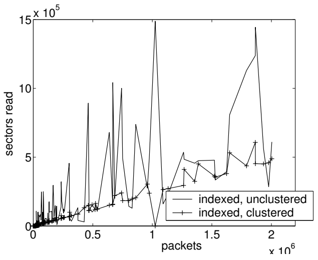


Fig. 10. Number of sectors read when executing the c-query for the BitTorrent trace.

no longer dependent on the number of connections but rather depends on the number of packets.

B. Measuring the Effectiveness of Caching

In order to measure the effect of caching, we executed several times the same c-query. First, the query was repeated 5 times in sequence for each connection in order to take advantage of available caching. During the second measurements, the query was executed once for all the connections and then the second time and so on until each query had been executed 5 times. In this way the caches got flushed between the subsequent executions of the same query. In order to focus on the effect of caching we excluded the first execution of each query from the measurements since only the subsequent ones benefit from the potential caching. As the results for both traces were similar, we only show those for the Gigabit trace.

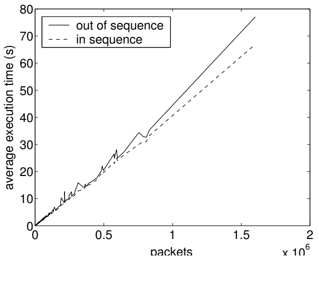


Fig. 11. Average execution time of the c-query for the Gigabit trace.

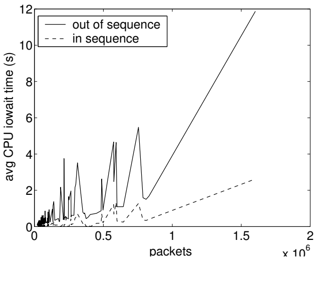


Fig. 12. Average CPU iowait time of the c-query for the Gigabit trace.

Figure 11 shows the evolution of the average execution time of the c-query in both cases. The difference between the cases with and without caching is minor: when caching, we gain at most 13% in the execution time. However, Figure 12 shows that the difference between the CPU iowait time is much more pronounced, which, together with the observations from Figure 11, means that the processes executing the c-query were most of the time CPU bound and not I/O bound. Indeed, a closer look revealed that, instead of waiting for pending I/O operations, the CPU spent most of the time doing user level computations (as opposed to kernel level computations), which suggests that the DBMS itself kept the CPU busy. We will continue this discussion at the end of this section.

C. The Impact of Parallel I/O: RAID Striping

We had the option to further minimize the I/O time through parallelizing those operations using a RAID system in striping mode. However, a glance at Figures 7 and 8 already shows that we could not expect a very significant speedup since the iowait times are already very low after indexing and clustering. Rough measurements confirmed that indeed the gain is marginal and, therefore, we decided not to use striping. Nevertheless, the fact that the performance of the c-query with the InTraBase is mostly CPU bound after indexing and clustering can not be generalized. With a faster CPU, multiple CPUs with parallel processing support from the DBMS, or slower disks the situation might not be similar and parallel I/O could improve significantly the performance.

D. DBMS as the Final Bottleneck

We have shown that after the proposed I/O optimizations, the performance of the c-query with the InTraBase is mostly CPU bound. On the average, the CPU spent approximately 84% of the time computing user level tasks with both of the traffic traces. We further investigated the origins of the main CPU activity by trying to exclude potential options.

First, we evaluated the effect of sorting (the c-query specified a chronological ordering of packets). We executed the c-query with and without the `ORDER BY` clause. The total execution times for the BitTorrent trace are plotted in Figure 13. We can see that the execution times without sorting are at most 15% shorter. Indeed, ordering the packets in main memory, especially as it is done while reading them from the disk, should not be the main bottleneck.

Second, we investigated the role of the size of the result set of the query on the execution time. We compared the

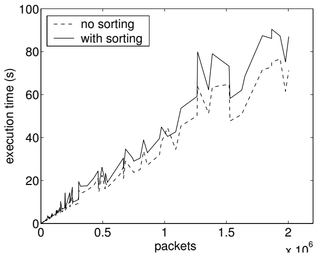


Fig. 13. Average execution time of the c-query with and without the ORDER BY clause for the BitTorrent trace.

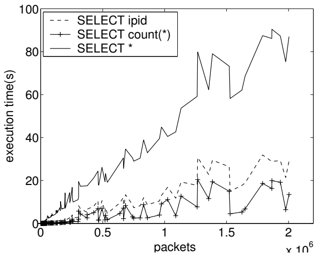


Fig. 14. Average execution times of the original c-query and a modified c-query that only counts packets for the BitTorrent trace.

execution times of the c-query and two modified c-queries: one which computed only the number of packets and another one which selected only the `ipid` attribute[6] as the query result, i.e. `SELECT count(*)` or `SELECT ipid` instead of `SELECT *`, respectively. As Figure 14 shows, the total execution times for both of the modified c-queries dropped up to 90% from the execution times of the original c-query. This result shows that the amount of CPU time spent on the user level computations is highly dependent on the result set size of the query. Thus, we have reason to assume that this CPU time goes to internal DBMS operations related to handling the result set tuples, e.g. transforming the physical storage format of the data into the logical format of the tuples, which we can not further optimize. This computational overhead is the price to pay for having the structured data and advanced querying facilities provided by the DBMS.

V. Conclusions

In this paper we have addressed the problem of how to efficiently perform packet-level traffic analysis using an off-the-shelf object-relational DBMS. As we have already detailed in [14], an approach based on a DBMS can address the major limitations in *management*, *analysis process cycle*, and *scalability* that we encounter with ad-hoc approaches using scripts and numerous specialized software tools. The challenge that remains is to perform this analysis efficiently. Such a system is useless if, in practice, the poor performance prohibits to take full advantage of the features to support complex analysis provided by the DBMS. We identified in this paper the steps of a typical network traffic analysis task which include a single common query: the c-query. We measured the performance of the c-query in the case of the InTraBase, which is a DBS for traffic analysis running on PostgreSQL under Linux. We showed that correct design choices reducing the cost of the I/O operations of the c-query bring great performance improvements. Moreover, we showed that the remaining performance bottleneck is due to the DBMS, and thus, is the price to pay for having structured data and advanced querying facilities.

We have shown with the performance measurements that the part involving querying with the DBMS of the typical analysis tasks execution scales approximately linearly with the number of packets queried. Furthermore, its total execution times for a packet trace of a few Gigabytes is from tens of minutes to a maximum of a few hours. However, the time needed for the initial processing of the packet trace to reach a stage where these analysis tasks can be executed (e.g. populating the base tables with the InTraBase, see [14]) may limit in practice the maximum size of a packet trace to be analyzed at a time. We have successfully used the InTraBase with packet traces of several tens of Gigabytes. For the type of analysis that we perform with the InTraBase, larger traces are generally not necessary. Nevertheless, based on these results, we would not recommend to use an off-the-shelf DBMS for very large scale packet-level measurement studies involving Terabytes of data at a time.

Acknowledgments

This work has been partly supported by France Telecom, project CRE-46126878.

The authors would like to thank Asbjørn Sannes and Karl-André Skevik for helping with the measurements and for maintaining the database server used for the InTraBase prototype.

References

[1] "Large-scale Monitoring of Broadband Internet Infrastructures (LOBSTER): http://www.ist-lobster.org/".

[2] "PostgreSQL: http://www.postgresql.org/".

[3] "Tcptrace: http://www.tcptrace.org/".

[4] M. Baldi, E. Baralis, and F. Risso, "Data Mining Techniques for Effective and Scalable Traffic Analysis", In *9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, May 2005.

[5] C.-M. Chen, M. Cochinwala, C. Petrone, M. Pucci, S. Samtani, P. Santa, and M. Mesiti, "Internet Traffic Warehouse", In *Proceedings of ACM SIGMOD*, pp. 550–558, 2000.

[6] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "The Gigascope Stream Database", *IEEE Data Eng. Bull., 26(1)*, 2003.

[7] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi, "Design and Deployment of a Passive Monitoring Infrastructure", In *Proceedings of Passive and Active Measurements(PAM)*, April 2001.

[8] H. Garcia-Molina, J. D. Ullman, and J. D. Widom, *Database Systems: The Complete Book*, Prentice Hall, 2001, ISBN 0130319953.

[9] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, G. Heber, and D. DeWitt, "Scientific Data Management in the Coming Decade", Technical Report, Microsoft Research, 2005.

[10] J. Jacobs and C. Humphrey, "Preserving Research Data", *Communications of the ACM*, 47(9):27–29, September 2004.

[11] N. Megiddo and D. S. Modha, "ARC: A Self-tuning, Low Overhead Replacement Cache", In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*, March 2003.

[12] V. Paxson, "Experiences with Internet Traffic Measurement and Analysis", Lecture at NTT Research, February 2004.

[13] D. Shasha and P. Bonnet, *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann Publishers, 2003, ISBN 1-55860-753-6.

[14] M. Siekkinen, E. W. Biersack, V. Goebel, T. Plagemann, and G. Urvoy-Keller, "InTraBase: Integrated Traffic Analysis Based on a Database Management System", In *Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, May 2005.

[15] M. Siekkinen, G. Urvoy-Keller, and E. W. Biersack, "On the Impact of Applications on TCP Transfers", Technical Report, Institut Eurecom, October 2005, http://www.eurecom.fr/~siekkine/pub/RR-05-147.pdf.

[16] M. Siekkinen, G. Urvoy-Keller, E. Biersack, and T. En-Najjary, "Root Cause Analysis for Long-Lived TCP Connections", In *Proceedings of CoNEXT*, October 2005.

[6] We tried also other types of attributes and obtained similar results.