Bachelorarbeit

How good can databases deal with Netflow data?

Ernesto Abarca Ortiz

Ingeniero Técnico en Informática de Gestión
September 20, 2011

Technische Universität Berlin

Deutsche Telekom Laboratories, FG INET
Research Group Prof. Anja Feldmann, Ph.D.

Supervisor: Prof. Anja Feldmann, Ph.D.
Advisors: Dr. rer. nat. Bernhard Ager
Dr. rer. nat. Fabian Schneider

Universidad Politécnica de Valencia

Escuela Técnica Superior de Ingeniería Informática
Department of Computer Engineering

Supervisor: Juan Carlos Ruiz, Ph.D.

Contents

INTRODUCCIÓN

Tiempo atrás, grupos como Pervasive Labs[1] han investigado en el campo de las tecnologias Netflow y las bases de datos, descubriendo su potencial y complejidad. El Lenguaje de Manipulación de Datos (DML) y el Lenguaje Estructurado de Consultas (SQL) para la obtencion de datos son probablemente algunas de las tecnologias mas utiles a nuestra disposición para el análisis de grandes cantidades de datos interrelacionados. También fueron constatados los grandes requerimientos de hardware necesarios para trabajar con dichos datos, haciendo el proyecto inalcanzable. Pero con el tiempo el hardware se abarata y se hace mas potente, y este cambio junto a algunas optimizaciones podrian hacer posible el proyecto.

Hoy en dia las redes gestionan enormes cantidades de trafico, y su diagnostico y analisis se vuelve mas dificil cada dia. Intentar guardar dicha informacion para su posterior analisis es impracticable. Esta es la razon de usar Netflow: recoger solo la informacion mas importante de cada conexion de datos.

Con Netflow recibimos estadisticas de routers y switches permitiendonos analizarlas mas tarde. Algunos de estos datos son el origen y el destino, la duracion y la hora de comienzo, ademas del tipo de datos y su tamaño. Almacenar esta información sigue sin ser fácil, y normalmente se ha realizado en archivos de formato propietario y con herramientas que dependen del vendedor.

Este trabajo evaluara varios sistemas de gestión de bases de datos (DBMS) como una alternativa a los archivos propietarios usando herramientas mas potentes y formatos mas abiertos, por ello elegiremos sistemas GNU/GPL en nuestra apuesta por el software libre.

Netflow tiene varias versiones, usaremos la mas comun para IPv4: la version 5. Y como bases de datos analizaremos: MySQL, PostgreSQL y SQLite con diferentes estructuras de datos y consultas. Para importar los datos usaremos la utilidad flow-export de las flow-utils con algunas mejoras.

INTRODUCTION

Some time ago, groups prior to us like Pervasive Labs[1] have previously researched into the field of Netflow and Databases discovering its strengths and weakness. The Data Manipulation Language (DML), further exposed, and Structured Query Language (SQL) for querying data are probably one of the most useful things that this technology can provide to us to analyse vast amounts of interrelated data. It was also shown its high-demanding requirements in terms of CPU power and storage space resources probably making it not worth to afford. But along last years while CPU power has greatly grown, storage space has been dramatically increased. This resource-cost change and the ability to transform data to a less CPU demanding resource can have better results or lower requirements.

Today's networks send and receive huge amounts of traffic. Network diagnosis and data tracking is becoming more and more difficult. Try to record and analyse that huge amount of data is almost impossible, and that's one of the main reasons to further develop the Netflow concept: aggregate the most important data from every connection in a monitored network.

With Netflow we receive statistics from routers and switches in near real time, allowing us to store it for further analysis or maybe even to react to problems in our network. Netflow will tell us statistics about every connection that crossed our network. Some of that information is the source and destination, the duration, time stamps and flow sizes. But even with that aggregation it is a vast amount of data and that is useful to store for future analysis. Storage and analysis of big amounts of data is complex and requires a lot of resources to process. Usually the storage and analysis of NetFlow data has been conducted with vendor specific tools and binary files to analyse them.

This work will evaluate the possibility to use common database management systems as an alternative solution because nowadays databases have greatly evolved and provide characteristics not available or not affordable in the old netflow-tools format. Some powerful characteristics of those systems usable in this work are: application/vendor data independency, cluster server data distribution and a powerful standardized data query framework.

There are many different database management systems, each one with its own data storage management system, retrieving-data language and methods to do it, and that could make this analysis too specific being a first approach to find, if it can be done, the way to store netflow data in a database system in a feasible way. As a first approach I will focus in SQL based DBMS to compare some database systems based in this technology.

As Netflow was designed with big networks in mind, this work focuses on storing huge amounts of netflow data and getting performance statistics to identify bottlenecks for solving or improving them. Also, as any technology, has its own limits and part of this work will be to identify them to know under which circumstances we can use it. It is unsure if this solution is better than the current situation so a comparison between different database management systems and the current solution should be done.

Netflow technology is evolving to new versions but we will focus on version 5 and IPv4 addresses as currently is the facto standard used in our institution. As a research group we prefer tools that are open to the community, we prefer free software and thus we support Linux O.S. And GNU/GPL database servers. Therefore, MySQL, PostgreSQL and SQLite were good candidates to participate in this work.

USE CASES OF NETFLOW/DB

Storing data in a database is not difficult, what can make it difficult is doing it in the best way for our needs and achieve the best performance, and to accomplish this we need to think about how is going to be used.

Following instructions I met with people from our department on the Measurement, Security and Routing areas to ask about how this work can help them and focusing this work on this target. Those are the common case uses of Netflow that should be considered on this work to be improved with databases.

- **Statistics**:
  - More than 70% of the requested 'features' about this work are related with getting statistics about network traffic.
  - Nearly all of them are statistics already found in network monitoring tools like Ntop, Cacti...
  - Statistics like Top Hosts, cumulative distributions (by prefix, AS, protocol, flow duration....).
  - General traffic statistics sorted, filtered and split along time laps in several ways.
  - Real time statistics were not requested on this area.
  - A useful feature would be to map AS numbers to ISP/networks.

- **Security Analysis:**
  - Looking into the contents of every single packet can be very tedious, not feasible, and even useless. Instead, taking a look to general network statistics can be really useful to find out the next step to follow or realise about a security problem such as connections from unexpected network areas, trojan traffic or any unexpected network traffic behaviour.
  - Monitoring connections can not be done by hand, but instead in an automated way by comparing traffic patterns along time, this monitoring is done with statistics and deviation parameters.
  - Netflow data can provide traffic patterns about P2P, worms, malware...
  - In this area, real time statistics or analysis can be needed for fast responsiveness against security threads.

- **BGP routing:**
  - As one of the main topics in our research group is communications between Autonomous Systems (AS). BGP routing protocol is of special interest and like other IP protocols Netflow will provide information about updates done between routers.
  - The aggregated data provided by netflows will not allow us to find specific BGP problems, but will provide useful information to find strange behaviours or to get a general overview about BGP traffic and create topology diagrams.
  - Netflows, reverse engineering and complex database queries can discover router policies on a remote network or, pointed by someone in our group, even who maid a BGP mistake and propagated it over our network.
  - As our group is really interested on this topic, correlating flows over time (among others) can be a very useful tool, and the use of databases can supply needs previously not covered by the use of old netflow analysing tools.

BACKGROUND

Netflow was introduced years ago by Cisco Systems for speeding up connections in routers and switches[2] with access lists by reorganizing the caches used to forward the traffic with the statistics collected from the device itself. This technique improves several parts of the device traffic management but also the collected data can be really useful for other meanings such network statistics or traffic surveillance.

Netflow architecture works as a side service to the network if possible not affecting communications as demonstrated in **Figure 1**. Configured routers and switches (exporters) on a network will collect data (connections) that crosses them, summarize (aggregate) it and send it to a specified host running a flow capturer (collector) that will store all received data. Later in off-line mode this data will be analysed by other tools.

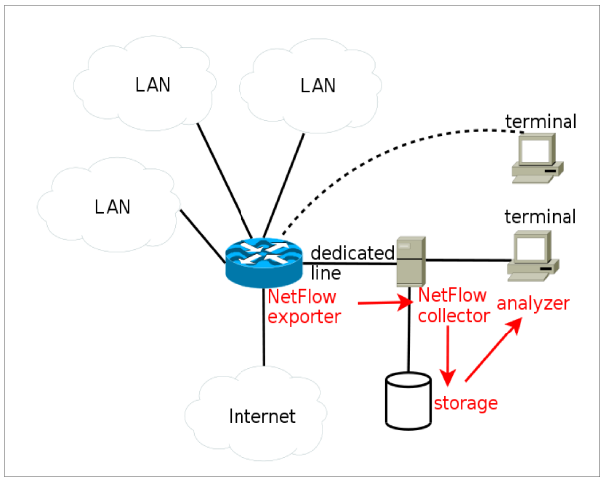


Figure 1: Netflow Architecture (src:wikipedia)

Connections in a network are also called flows, and are identified by those fields together: *source IP, source port, destination IP* and *destination port,* and even if connections are bi-directional, for Netflow they are considered one-way flows and thus we will have two flows for each connection.

Netflow works in the following manner: When the first packet of a new connection crosses the switch or router a new netflow record is created in the device's cache to track all information related to it like: number of packets, bytes sent, source and destination *Autonomous System* number, time stamp of the first and last packet, source and destination IP Address/Port and network mask, incoming/outcoming interface, *Internet Protocol* type used and other fields.

There are some conditions that will consider a flow as finished. Obviously one of them is the *end of transmission* packet or receiving a *reset* packet. For broken connections can be the *idle time* condition, and for long connections a *timeout* can be configured. After one of this conditions is met the flow data is marked as ready to be sent.

When some of those flows are ready the device puts together several of them, adds a header with common information to create the full netflow packet and sends them all together to the configured destination for being further processed.

There have been several versions of the netflow protocol along the time, having thus, differences. The first version (v1) was restricted to IPv4 and neither IP mask or AS numbers, it is a very old version now obsolete. Version 2, 3 and 4 were never released to the public domain and only used internally by Cisco. Version 5 is the most commonly used version and the one we will use, further details following. Version 6 is not any more used and only had extra information about the protocol encapsulation. Version 7 is like version 5 with a source router field. The 8th version had also the same information as version 5 but with several ways of data aggregation, and finally version 9 and 10 are based on templates, allowing more flexibility for big networks. Version 10 is actually being ratified as an IETF standard.

The most widely used packet is version 5, having two parts: a general flow header format as seen in **Table 1** referring to up to 30 flow records, each specific flow data recorded as in **Table 2** field format.

Table 1: Netflow version 5 original header packet format

Bytes	Field name	Description
0-1	version	Netflow export format version number
2-3	count	Number of flows exported in this packet (1-30)
4-7	sys_uptime	Current time in milliseconds since the export device booted
8-11	unix_secs	Current count of seconds since 0000 UTC 1970 (Epoch)
12-15	unix_nsecs	Residual nanoseconds since 0000 UTC 1970
16-19	flow_sequence	Sequence counter of total flows seen
20	engine_type	Type of flow-switching engine
21	engine_id	Slot number of the flow-switching engine
22-23	sampling_interval	First two bits hold the sampling mode; remaining 14 bits hold value of sampling interval

Table 2: Netflow version 5 original flow record format

Bytes	Field name	Description
0-3	srcaddr	Source IP address
4-7	dstaddr	Destination IP address
8-11	nexthop	IP address of next hop router
12-13	input	SNMP index of input interface
14-15	output	SNMP index of output interface
16-19	dPkts	Packets in the flow
20-23	dOctets	Total number of Layer 3 bytes in the packets of the flow
24-27	first	SysUptime at start of flow
28-31	last	SysUptime at the time the last packet of the flow was received
32-33	srcport	TCP/UDP source port number or equivalent
34-35	dstport	TCP/UDP destination port number or equivalent
36	pad1	Unused (zero) bytes
37	tcp_flags	Cumulative OR of TCP flags
38	prot	IP protocol type (for example, TCP = 6; UDP = 17)
39	tos	IP type of service (ToS)
40-41	src_as	Autonomous system number of the source, either origin or peer
42-43	dst_as	Autonomous system number of the destination, either origin or peer
44	src_mask	Source address prefix mask bits
45	dst_mask	Destination address prefix mask bits
46-47	pad2	Unused (zero) bytes

A database can be just a human-readable file or a group of binary files managed by a whole system, but what they have in common is that they store very structured data together, and usually, big amounts of them.

Along time, data has been stored in different ways, many years ago nearly human readable plain text file were used to store data, that data requires as much storage space as we see: one byte for each character, including spaces and other hidden characters. The following text is structured data in one of the many possible plain text formats, this is Comma Separated Values (CSV):

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""","",5000.00
```

This data requires one byte for each character, and they require to be converted into internal computer binary representation to be processed, that needs more resources. But there are other ways to store the same data, some of them are not human readable, one of them is called *binary data*. As an example, the number *1997* here requires 4 bytes and needs a binary conversion, but stored as binary data can require only 1 byte and no need to do the binary conversion.

As binary data is not supposed to be read by a human, it can be stored depending on technical reasons. Some ways can be in a smaller compressed form that saves us space or in a long format if it improves performance or makes it easier to modify data: using the right choice for our needs will make the difference. Usually the following resources are implied when working with data: CPU, Input/Output and storage space. Each data type has a different impact on those resources but they work together. Thus, using the right data type with the right resource impact combination will help to get the best performance.

When first database systems appeared they started to store data in binary format, requiring less time to convert and process data and adding indexes to be able to find quicker the solicited data. They used to provide a vendor specific interface to access data, that allowed programmers to spend the time in developing applications instead of programming lower levels data management. There are different kinds of Database Management Systems (DBMS), most work with a concept called *Tables*. A table has a designed data structure allowing to store specific entity (event) data, and by having several tables we can store all data we are going to use.

A single datum is an event from our world being stored, and as our world, everything is related and thus the data too. As databases were growing data started to be duplicated and more correlated and that raised a problem: having same data duplicated requires keeping up to date every single instance, requiring more resources. Next step in database systems was to correlate stored data between tables, this allowed to have less duplicated data and by means of strict *relationships* data integrity and consistency reached a new level. Next level was to allow databases to have his own *language* to do different tasks, this is called *Data Manipulation Language* (DML). DML is a structured language to create and define the whole database, to alter it and to fully access and manipulate data contained on it.

As explained, *relational databases* are a very effective form to store and keep organized well structured data with the possibility to 'connect' it to related information. Today, information is power and storing and accessing to it is really important. Vendor specific methods and languages to access data finally became a problem as migrating data from system to system was really difficult. A new common method to access data emerged several years ago as part of a DML to help: *Structured Query Language* (SQL) and most important database systems support this language today, though with different versions and functionalities. All of them conform at least to one of the old wide spread versions: SQL-92 or SQL-2000. As a bottom line can be said that “When having *big amounts of data the best to analyse it is to have the most powerful tools”*.

APPROACH

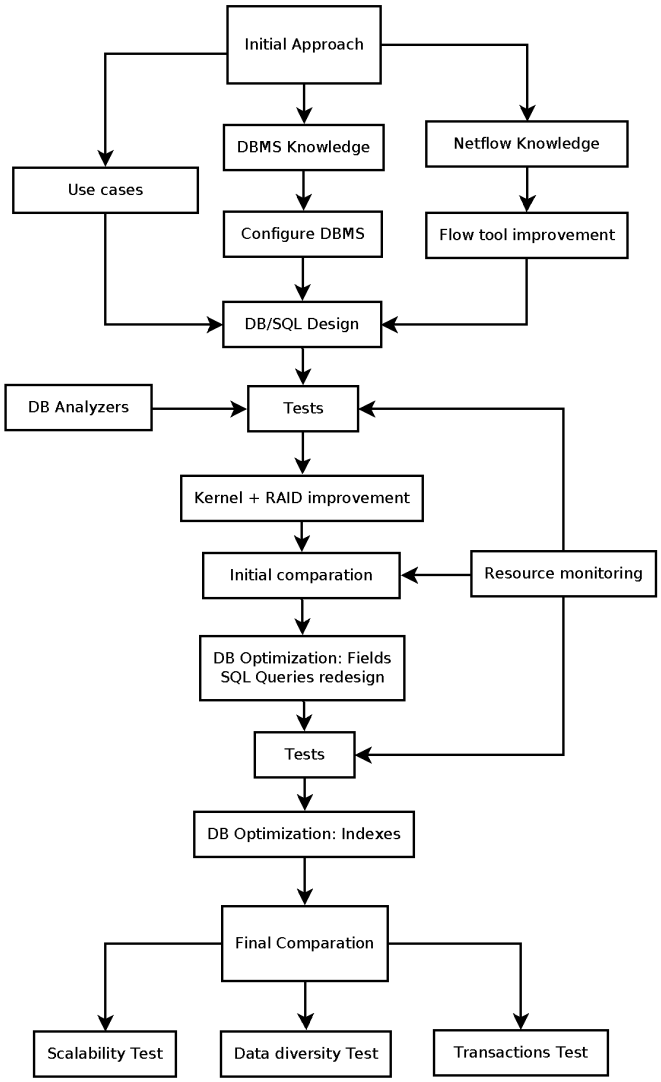


Figure 2: First approach elected in this work

For an initial approach we will join three important factors together: Database knowledge, Netflow knowledge and our specific needs as a networks research group. All together will allow us to design a system that should be able to answer our requirements.

Database analysers and monitoring of resources will conduct us along all our trajectory on the right path. While an initial test will verify that we are in the right starting point, our knowledge about networks and databases will lead us from initial data optimization to database optimization.

Having discovered along the way new possibilities that will allow us to test new ideas. Ideas that will direct us to discard some of them and point us to a new beginning with a deep knowledge about this solution to compare it with new ones.

Diagram shown in **Figure 2** shows the process followed as an initial approach, having a single table in the DB to store all data.

Storing Netflow onto a database can have another great advantage, if we can use a standard like SQL to access data, we will be able to choose between different database systems and be vendor-free. This common method to access data to/from the database will also allow us to use very similar methods to analyse and compare them, obtaining trustable results. This common method is evolving to a more powerful and more flexible language allowing us to retrieve results easier and faster. But each vendor tries to improve its own database system creating some differences in the language by means of optimizations.

Those optimizations to the different aspects to the database system sometimes can make a big difference but also some incompatibilities, we will leave them out of our scope for this work to be implemented in the final implementation. As even using the same SQL language, the implementation of the DB can be different for each DBMS and field type, a common design will be created and another one with some specific characteristics to know which one is performs better.

The one-table design has been chosen as the first approach for several reasons like: easy record insertion, existing flow export tool, no need to lookup master keys... That simplicity can be a good way to deal with some hundred millions of flow records each day, as even with only one table many optimizations can be done. Other ways to store the data into a relational database system can be:

- Using also only one table:
  By means of hash functions, create a shorter unique primary key with the AS-IP Addr-Mask-Protocol fields to store it instead of those fields, requiring less space and disk I/O. A shorter and unique key will allow us to fetch more efficiently required records but the trade-off of this solution is the cost of increasing query complexity, this will not allow a person to create the queries by hand, but the use of functions or a user interface can help on this problem.

- Using several tables:
  With two tables: A master table for the fields who identify a flow or with the most important fields (AS-IP Addr-Mask-Protocol) plus an unique identifier if needed, and a secondary/slave table for all the other data. By having a 1:N (master-slave) relationship, the most repetitive data will be written only once in the master table, this will save I/O to disk probably making it faster to save flow records.
  The trade-offs for this solution, is that a unique identifier is needed to correlate data from both tables and the need to rewrite the flow-export tool. Looking up for this identifier in the master key each time we receive a new flow can be very slow, a possible solution for this problem would be to implement a cache on the flow-export tool having last ten to fifty thousand of stored identifiers.

  Three or more tables: Following this idea, data can be split even in more tables, e.g. storing in a third table data not usually requested as input/output SNMP interface or Type of Service. Another table can be used to store common fields for querying data, the destination address for the next hop can be a good one, source and destination AS may be also candidates. There are many possible combinations and its performance can depend also on network topology or traffic pattern, this would require a full new research that can be taken after the present one.

- Using temporary tables to summarize information as it is being received and store only needed information in a permanent table every day/hour/minute, making smaller the problem of storing and retrieving data. The trade-off is quite obvious, this solution looses connections details and may be not acceptable to us.

CHALLENGES

Different DBMS have different possibilities, each one with its own characteristics, and some of those possibilities can change results substantially, as this work is a first approach we will leave the DBMS mainly with the default out-of-the-box options and concentrate on our work so further research after this one can be done in the optimization of servers if needed.

In any system and at any level, buffers are a common technique to improve performance, and DBMS also have this option heavily effecting performance. But enabling this option has a side effect: if e.g. power supply fails, some data can be lost; but this side effect can be solved by several ways. As an example, this option is enabled by default on MySQL[3] ('*flush*' option) but not on PostgreSQL[4] ('*fsync*' option) or SQLite[5] ('*pragma synchronous*' option). To equally compare DBMS this behaviour should be taken in account, and as there are many ways to solve inconveniences, we will enable buffers to find out the maximum performance reachable. As using buffers can have non-realistic behaviour when running tests, care should be take to avoid test interaction by flushing data still in the buffers to the disk.

After the initial DB design and looking to *key_reads/writes* from MySQL '*show global status*' and tools like *sqlite3_analyzer* for field type and row sizes overhead, the I/O performance was detected as the greatest bottleneck as it is obvious by the huge quantity of data to be stored. As NetFlow packets were never in mind to be stored in a DB, disk space and thus I/O performance is wasted depending on DB field size (32/64 bit), data format (IP Addresses as plain text/number), useless fields for our purposes or data that is duplicated like time stamp fields.

Filtering large amounts of data can be a resource-expensive operation, databases have the possibility to use indexes for faster data access, but this option can require even more I/O access making it slower rather than faster. Most probably the use of indexes is not worth for us as usually we will retrieve much more data than the low percentage the indexes where designed for. But as the characteristics of our data is mainly WORM (write once, read many) there can be a chance to be worth. Also as indexes require more data to be written to the disk, this write operation should be as fast as possible if we want to use it in a real time network-to-database packet dump without using the old flow-tools method as an intermediate step. At the end, in case real time indexing is not possible due to performance issues, offline indexing such as delayed distributed indexing can be an option, for this reason we will research on indexing usage anyway. Indexes can be created with different field types and combinations of them, some tests should be done to test the feasibility of this feature or to better know how to design indexes. Through data to be stored is only formed by numbers and probably they will have a smaller data diversity than one formed by the full alphabet, this data diversity can really affect to index creation and we should be aware of this.

As we want to compare our new DB methodology with the old flow-tools methodology we should compare them with the same task, method and obtaining the same results. Every DBMS has several methods/libraries to retrieve the data, and each one can have its own architecture, behaviour and performance. Chosen DBMS have in common a standard, the Structured Query Language (SQL) to access the data, this method allows us to fetch the data in the same way for any of those databases making results comparable. This query language is very powerful and even being a standard, each DBMS has its own optimizations that really can affect the performance. We need to create as much generic queries as we can for reliable results, leaving the in-DB optimizations for further research.

RESOLUTION

To equally compare all DBMS, option to use buffers (sometimes known as: disable buffer synchronization) has been enabled on all of them. In case of MySQL the environment *flush* variable was set to OFF to get this behaviour, while in PostgreSQL configuration file *fsync* option was set to OFF and in SQLite the environment option *Pragma synchronous* was configured to 0 (OFF)

Disk buffers on system memory have a major impact on timing results. As I/O is the slowest part of our system, having that data already in a fast memory will distort results. To achieve comparable and repeatable results, we clear those buffers between experiments as explained in the *Testbed Description* part of the *Dataset and Testbed* section.

But even enabling buffers, I/O performance was still too slow not allowing us to store our dataset neither being faster enough for a medium-sized network. Resource monitoring was used to find bottlenecks and solve them up to an acceptable point. Commands such *dstat, iostat* and tools like *MRTG* were used to monitor CPU, hard disk I/O, free space used and RAM memory usage.

MONITORING RESOURCES

While *dstat* was used for real time monitoring over all resources and processors to fine tune other monitoring tools, *iostat* tool is only used for I/O stats. Last one is called from *flow-export* every 100.000 records are stored into the database to fetch the input/output KB read/written from/to the hard disk database partition as follows: `iostat -kd /dev/sda3 | grep "sda3" | awk '{printf \"%s\\t%s\\n\", $5, $6}'` This data joined with time stamps was used to create the time-spent/data-written over saved records graphs later showed.

MRTG was configured to supervise, among others, the following system properties:

- User/Kernel Raw CPU usage to analyse *flow-export*, DBMS and kernel behaviour.
- CPU I/O RawWait to improve I/O speed operations
- Data read/written from/to the database storage partition to control data I/O and correlate it over time spent.
- Memory/cache usage to control memory usage and cache behaviour.

On the following, MRTG graphs will show system behaviour in general tests, while in the *Results* section they will show on a per-test basis the bottlenecks found and how them affect to the system, proposing solutions to solve them.

At the very beginning, a low level bug was found slowing down I/O access, the SCSI controller is assumed by our Routerlab administrators to be the reason of this behaviour. Without this fix, no one database was able to deal with the big amounts of data we were in the needed to store. By means of comparison of different Loadgen machines, kernel versions, SCSI modules loaded and system behaviour it was found that using different SCSI kernel modules, the same kernel had a great performance increase. The kernel options are '*Fusion MPT ScsiHost drivers for SAS*' and the wrong modules being loaded correspond to *mptbase*, *mptsas* and *mptscsih*, through they officially support our LSI SAS 1068E card they do not seem to work very well. Even more, at the moment of writing the *Conclusion* section another optimization was found, while all tests have been done running kernel 2.6.30.10 running the new MPT2SAS modules, removing them from the kernel significantly reduced the time to perform some indexed tests.

Taking as a base a default[I] image being used at the laboratory and after running a sample test several times, **Figures 3 and 4** revealed the first bottleneck. System has a high CPU I/O wait time:

[I] Debian Lenny 32bit running a 2.6.18-6 686 kernel.

under-using CPU power for the DBMS running as a system process and for the *flow-export* tool running as a user process. Further research demonstrated the wrong loaded kernel module for the storage controller as the reason of this penalty. Graphs show in the first 24h the problem, while in the last 10h the problem is solved.

This behaviour caused the high-performance system to be able to write only around 7 Mb/sec to the hard disk as seen at the beginning in **Figure 5.** The same problem also leaded the module to send about four times more data to the hard disk, this caused the tests to require much more time to finish as seen in the peaks difference between the first 24h and the last 10h shown in the graphs. In **Figure 6** we can appreciate the kernel cache usage and behaviour and verify that memory is being released. Once solved, the system was able to deal with the same task four times faster and not being overloaded in I/O operations.

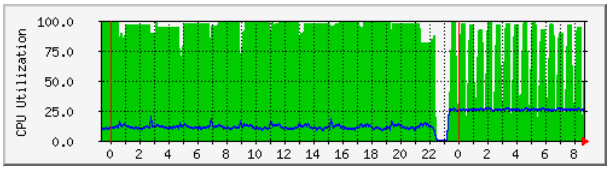


Figure 3: CPU: I/O RawWait(green) - System processes(blue)

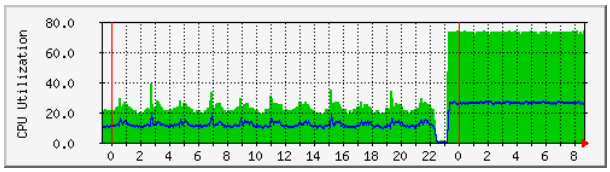


Figure 4: CPU usage processes: User(green) - System(blue)

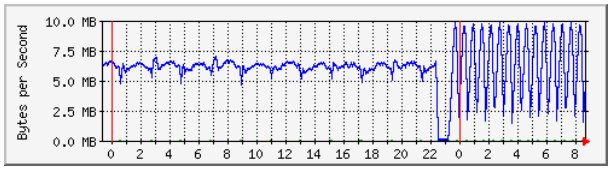


Figure 5: /mnt/databases Read(green) - Write(blue)

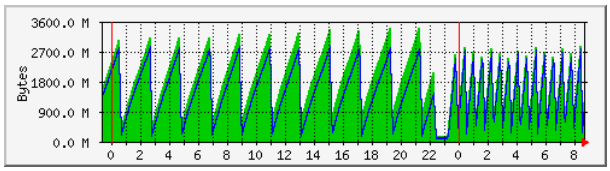


Figure 6: Memory Total/cached

FIELD TYPES

As seen above, I/O activity heavily depend on the quantity and type of data, one of the main topics of this study will be this topic. It is very common in computer science to store timestamps as the number of seconds since 1970-January-1st usually called Unix Epoch, the advantage is that it fits just as an integer number using 32 bits (4 bytes) while all time-stamp fields on databases require 8-12 bytes, requiring thus more data to be read/written. E.g.: `Fri Sep 17 2010 20:40:43` will be converted to number `1284748843`. Database DateTime field type requires 8-12 bytes because it allows to store more data and more precise than Epoch field like: fractions of a second, time zone, wider range of dates, or even time intervals. Epoch timestamps can only store dates from year 1970 until 2038, one-second precision and no time zone information, while a standard DateTime field can store dates from year 4713 BC until 5874897 AD with microsecond precision and time zone information.

For IP addresses there are two common ways to store it: as normal text requiring from 7 bytes up to 15 or with a simple formula converting it to a number[6], requiring only 4 bytes. Doing this operation is safe as the maximum IPv4 Address value: 255.255.255.255 will be converted to: 4.294.967.295 that perfectly fits in a 4 byte unsigned integer field. The unsigned (only positive numbers) characteristic is very important as without this the maximum value will be only 2.147.483.648 and it can not be stored and will require more bytes to do it.

Depending on the DB there is a third option, to use a specific field type like PostgreSQL INET field (12 bytes) for this data that usually requires less bytes than the plain text version and also offers specific functions and syntax but only small speed improvement. The conversion is, having a dotted IP Address: `aaa.bbb.ccc.ddd` and the formula: `(aaa*256³)+(bbb*256²)+(ccc*256)+(ddd)` the IP Address: `192.168.55.89` will be converted to: `3232249689`. The side effect of storing timestamps or IP Address as integer numbers is that we will need to convert the data and take care about how is being stored at the time of designing the SQL query as it is not the same comparing “> 2008-05-15” as

text/date than “> 1210802400” as an epoch number.

Results from this work will help to determine the optimal way to store data in the future, but for now we should adapt original Netflow information to our needs. Usual data sent in Netflow packets has two main disadvantages for our research: nearly useless or not needed fields, and dependency in time stamp fields between Netflow *header packet* and *flow records*. To get into the worst-case of storage requirements and data manipulation we will try to store nearly all fields and only a few unneeded fields will be ignored. In this study, original and combined chosen fields written to the database are shown in **Table 3** while in **Table 4** ignored fields and the reason are shown.

Table 3: Fields stored in our database

Field name	Description
src_exporter	Source IP address of router/switch exporting flows
secs_flowstart	Timestamp when flow started
secs_flowend	Timestamp when flow finished
secs_export	Timestamp when flow was saved to the DB. Alternative timestamp
src_addr	Source IP address of flow
src_mask	Source address prefix mask bits
src_port	TCP/UDP source port number or equivalent
src_as	Autonomous system number of the source, either origin or peer
dst_addr	Destination IP address
dst_mask	Destination address prefix mask bits
dst_port	TCP/UDP destination port number or equivalent
dst_as	Autonomous system number of the destination, either origin or peer
input_if	SNMP number for input interface at the exporter router/switch
output_if	SNMP number for output interface at the exporter router/switch
next_hop	IP address of next hop router
num_packets	Packets in the flow
num_bytes	Total number of Layer 3 bytes in the packets of the flow
ip_prot	IP protocol type
tcp_flags	Cumulative OR of TCP flags
ip_tos	IP type of service (ToS)

*Table 4: Fields **not** stored in our database*

Field name	Reason
version	Unneeded: Always will be version 5 in our experiment.
count	Unneeded: Each flow is a record.
sys_uptime	Combined: No need of nanosecond precision and dependency between fields will require more pro-record calculations.
unix_secs	
unix_nsecs	
flow_sequence	Unneeded data in a pro-flow record database.
engine_type	Unneeded data.
engine_id	Unneeded data.
sampling_interval	Unneeded data in a pro-flow record database.
first	Combined: Added to Unix timestamps above we calculate secs_flowstart and secs_flowend
last	
pad1 pad2	Ending optimization: Perhaps in final implementation a data padding pro record can be worth to align to storage sector size.

Depending on our needs some fields can be removed, for example: *input_if* and *output_if* probably they will be not frequently used or required. Field *src_exporter*, has been added from the IP layer packet as it can be really useful to filter data from a big network only with this field. Netflow protocol uses the local exporter device time and date to know when connections have started and when finished, if this time-stamp is wrong or not well synchronized with other devices that will result in data misplaced on time. Our own field *secs_export* is used to correct this or use it as a reference to fix wrongly exported data, but in the case we are sure about time synchronization on all devices, this field can also be removed.

To be able to compare the performance between different kind of fields and sizes, database was redesigned to use also alternative field types to store DateTime and IP addresses creating thus two variants: the first one only with Int32 fields being *generic* to all databases and another one with database specific fields like PostgreSQL:INET or MySQL:DateTime, sometimes referenced here as *plaintext* as they does not require data conversion by us on SQL queries. The difference between data field sizes and calculated row sizes can be seen in **Table 5**.

Table 5: Generic vs Specific field type DB table format

FIELD	**DATABASE SYSTEM**					
	MySql *		PostgreSQL		SQLite3 **	
	Generic	**Specific**	**Generic**	**Specific**	**Generic**	**Specific**
IP Addr fields (x3)	**Int32 ***	**Text (15b)**	**Int64**	**Inet (12b)**	**Int**	**Text (var**)**
Timestamps (x3)	**Int32 ***	**DateTime (8b)**	**Int64**	**Timestamp (8b)**	**Int**	**Text (var**)**
src & dst mask	Int8	Int8	Int16	Int16	Int	Int
src & dst port	Int16	Int16	Int32	Int32	Int	Int
src & dst as	Int16	Int16	Int32	Int32	Int	Int
input/output if	Int16	Int16	Int32	Int16	Int	Int
num_packets	Int32	Int32	Int64	Int64	Int	Int
num_bytes	Int32	Int32	Int64	Int64	Int	Int
ip_prot	Int8	Int8	Int16	Int16	Int	Int
tcp_flags	Int8	Int8	Int16	Int16	Int	Int
ip_tos	Int8	Int8	Int16	Int16	Int	Int
Row size:	49 bytes	94 bytes	98 bytes	110 bytes	(variable)	(variable)

* MySql supports unsigned numeric types, allowing us to use smaller types.
** SQLite3 adapts automatically field type and size to the received value.

Specified row size indicates in theory how much space will be required by each flow record in the table giving us an idea about the difference of storage requirements between databases. It can be easily appreciated how PostgreSQL requires twice the space in the 'generic field' table but only a bit more in the '*plaintext* field' table mainly because specific time/date and IP Addresses fields are practically identical.

Field sizes were chosen depending on the data they will contain, electing the field size that will be able to store the maximum value by the netflow packet field, or the converted value. As MySQL allows unsigned numbers: IP mask fields will store a value from 0-32 that will fit in an INT(8 bits) that allows the 0-255 range, similarly it can be applied to the IP protocol field, TCP flags and Interface number fields, port numbers are on the range 0-65535 and requires a 16 bit unsigned integer type (INT16) while big values fields like num_packets and num_bytes (with values up to 2^{32}) require a 4 bytes field (INT32). In the 'generic field' table as IP Addresses and timestamps are saved as 32 bit numbers they require a full INT32 4 bytes field type, but in the '*plaintext*' version specific database types

are used requiring different byte amounts.

As MySQL has unsigned types but PostgreSQL has not, field types in PostgreSQL will need to be bigger, exactly twice the size of MySQL fields. And if PostgreSQL has no internal optimization to not store those unused bytes it will 'waste' on disk half of the reserved storage space. SQLite version 3 is a small desktop DB with no client/server architecture intended for small and medium datasets doesn't have many features presents in any other DB, one of them is field type specification. SQLite detects data type every time it access data, and stores it with variable size fields depending on the value received.

INDEXING

It is a fact that field type heavily affects field indexing, and as in this study we have different field types an index analysis based on the usual statistics required for this kind of data will be done. Our case uses reveal that requirements to create network data statistics mainly need to correlate or filter data based on the following fields: AS numbers, IP Address, IP Ports and IP exporter address. They also need to filter data by flow timestamp to specific ranges and exported timestamp to database in case of wrong timestamp synchronization between routers should also be considered. This requirements are also very common in any network monitoring tools like Cacti, Ntop and others. Based on typical fields to be sorted and filtered, in **Table 6** are shown three different proposed groups for index creation, called *0*, *A* and *B* with different combinations of single and compound index fields.

Table 6: Group Indexes: simple and compound fields

Group 0	Group A	Group B
IP Addr exporter	IP Addr exporter	IP Addr exporter
Flow exported time	Flow exported time	Flow exported time
Flow Start time	Flow Start time + Flow End time	Flow Start time + Flow End time
Flow End time	Src AS + Src IP Addr	Src IP Addr + Src IP Port
Src AS	Dst AS + Dst IP Addr	Dst IP Addr + Dst IP Port
Dst AS		Src AS + Dst AS
Src IP Addr		
Dst IP Addr		

Depending on several conditions and the specific SQL query received, databases will use or not indexed fields. Some of the conditions they focus on are: type of field, possible speed improvement based on internal statistics, field included in the filtering part of the query, simple or compound field and so on. As it is very common to filter by flow timestamps and IP address exporter all index groups have those fields. While Group 0 has only simple fields that should be used easier by the database engine than compound fields, Group A and B have compound fields for the commonly fields used together, joined in two different combinations.

Common denominators are:

All groups: Use indexes for IP address exporter and flow timestamps
Group 0: Use simple field indexing only.
Group A: Compound fields by Src or Dst AS+IP Address
Group B: Compound fields by Src and Dst IP Addr+IP Port and Src+Dst AS

DATA QUERY

To test and compare the *netflow-tools*[7] library and selected database systems we will design

similar experiments (like data aggregation or statistics generation) for both methods, those experiments will be based on our real needs as stated in section ***Use cases of Netflow/DB***. As experiments should be as similar as possible, SQL queries will be simplified and created conforming to a generic SQL syntax accepted by all of them and leaving the use of specific functions and vendor specific extensions out of this work. Only mandatory conversions needed to compare data for different field types have been included in the queries and general statistic counters usually included in *flow-tools* reports will be added to SQL queries to force the DB to do the same amount of work as *flow-tools* are doing. While SQL is flexible and powerful, *netflow-tools* is not so flexible and we are limited to use existing reports or combine and filter some of them. Two reports were selected with different characteristics to compare filtering, indexing and data aggregation:

Query A: Number of flows every 10 minute time windows along all the stored data. As an example the query for MySQL database, storing DateTime fields as Integers:

```
SELECT from_unixtime(floor(fl_secs_flowstart/600)*600) as timeslide, count(*) as flow_count
FROM   flows
GROUP BY timeslide
ORDER BY timeslide;
```

This query returns only 145 records, will not get profit of indexes, process all records and doing slightly aggregation by one restricted field with low data diversity. Partial result:

Time-window-start	flows	Time-window-start	flows
2003-05-28 00:00:00	100479	2003-05-28 00:40:00	96408
2003-05-28 00:10:00	98517	2003-05-28 00:50:00	94502
2003-05-28 00:20:00	97923	2003-05-28 01:00:00	105628
2003-05-28 00:30:00	99385	2003-05-28 01:10:00	95985

Query B: Number of connections and general statistics between 10:00-20:00; aggregated by Src IP Addr, Dst IP Addr, Src Port, Dst Port, IP protocol number and IP Type of Service. The query for MySQL database, storing IP Addresses as integers is as follows:

```
SELECT inet_ntoa(fl_srcaddr) AS SrcIP, inet_ntoa(fl_dstaddr) AS DstIP, fl_srcport AS Sport,
     fl_dstport AS DPort, fl_ipprot AS Prot, fl_iptos AS ToS, count(*) AS flows,
     sum(fl_numbytes) AS octets, sum(fl_numpackets) AS packets
FROM flows f
WHERE fl_secs_flowstart >= unix_timestamp('2003-05-28 10:00:00') and
      fl_secs_flowend   <= unix_timestamp('2003-05-28 20:00:00')
GROUP BY fl_srcaddr, fl_dstaddr, fl_srcport, fl_dstport, fl_ipprot, fl_iptos
ORDER BY octets DESC;
```

This query returns near 2.9 million records because we are aggregating by many fields with a diversity index much higher than the previous one. This query will try to take advantage of flow timestamps filtering to don't process all records and a heavy use of aggregation. Partial result:

src-address	dst-address	SrcPort	DstPort	Prot	ToS	flows	octets	packets
128.109.192.0	131.96.0.0	37853	119	6	0	575	449716694	303600
140.142.9.47	233.0.73.20	1026	8000	17	0	590	288604928	206564
137.78.56.0	140.90.192.0	22	34546	6	0	589	192549428	131225
128.109.40.0	198.202.120.0	32770	32774	17	0	8	102342405	68365
128.59.31.169	224.2.211.27	61552	61552	17	0	587	78807899	65264

SQL TRANSACTIONS

Index creation was found to be very slow and some techniques were proved to improve this task. This task seems to have two important steps based on database behaviour: index in memory creation and index on disk storage. In-memory creation requires mainly CPU power, showing a bottleneck by using only one CPU at a time because tested database systems are not multi-cpu versions for this task. The

second step, index on disk storage, can be optimized with any kind of buffer usage, this buffer can be implemented by disabling the sync to disk option as shown at the beginning, using SQL transactions to send together several thousands of data insertions so indexes can be processed at a time or by system and hardware improvements.

With the use of SQL transactions, relational databases can implement atomic operations, this technique allows to send complex and/or multiple queries to be executed together, assuring that all of them will be successfully executed or every modified data will be reverted in case it wasn't. In our situation the DB will process all the queries included in the same transaction together, probably creating also the indexes at the same time without requiring slow I/O disk transfers and greatly reducing the 'data diversity' problem as stated later. All analysed DBMS support at least simple transactions, we forced the use of them through source code modifications on the *flow-export* tool.

An easier hardware configuration for faster I/O was to reconfigure the storage RAID system, a RAID system are a group of hard disks that can be configured in several ways to provide faster speed performance, data duplicity to resist hardware failures or both**.** A really easy and simple improvement was to convert the Sun Microsystems RAID 1E into a RAID 0. While the first one provides data reliability, the second one provides a much higher I/O transfer ratio. The later was chosen to find out the maximum reachable speed.

DATA GENERATION

Only when using indexes, tests using different data sources like real data and randomly generated data showed a different I/O behaviour. Data analysis reported that I/O speed highly depends on data being processed, *data diversity* or differences seen between in-data was the reason for this behaviour: as inserted data has bigger differences with the already stored data, time needed for index creation increases. Flow-tools have a non-realistic netflow generator, the algorithm changes over the time but as for version 0.67 the flow generation just increases by one every single field in each flow. This data is not realistic at all, its characteristics are: huge amounts of unrelated flows without aggregation data and without similarities. Generated data looks like the following:

```
# flow-gen -n 100 |flow-print
srcIP            dstIP            prot  srcPort  dstPort  octets      packets
[...]
0.0.0.88         255.255.0.88     17    88       65368    89          89
0.0.0.89         255.255.0.89     17    89       65369    90          90
0.0.0.90         255.255.0.90     17    90       65370    91          91
[...]
```

Other netflow generation tools were tested showing some kind of realistic traffic but after some more testing with the *flow-gen* generated data and some strange behaviour at the end of our dataset I realized about how much data diversity can affect results. I thought it was a *must* to test with that kind of data to discover database scalability and behaviour on worst conditions. This kind of data can be seen as a very big network; even with millions of flows data will be poorly related and data duplicity will be very small. Along tests on databases with indexes, this data showed a very different behaviour in CPU usage and I/O throughput, being much higher than on our real data tests, through on tests without indexes results were not affected. This test is not realistic and not in-deep verified but should teach us to take care about the traffic pattern (usually network size, design and usage) of the expected target network where this work will be deployed in case of using indexes.

FLOW-TOOLS IMPROVEMENTS

Finally, to support all designed tests, *flow-export* tool was modified several times to fit our new requirements. Some of those modifications were:

- Export data to different database systems: Added SQLite3 and PostgreSQL support improved.

- Field type conversion: DateTime and IP Addresses to integer conversion.
- Unification and correction of timestamp fields: Netflow header packet and flow records.
- Data insertion statistics gathering: Disk I/O Kb transferred pro 100.000 SQL insertions.
- Transaction control: Auto mode, disabled mode and specified records pro transaction mode.

The new look that *flow-export* tool has after these modifications is the following:

```
./flow-export-v7 -h
Usage: flow-export-v7 [-l|-s] [-t numflows] [-h] [-d debug_level] [-f format] [-m mask_fields] -u
[database URI]
        -l         Translate IP Addresses to long integer (Incompatible with -s)
        -s         Use ' as SQL separator instead of " for [Inet/PGSQL] field type (Incompatible with -l)
        -t flows   Specify how many flows pro SQL transaction will be sent (default: 1000)

        Note: Transactions are ON and in AUTO mode unless you use   -t 0   to disable them.
        Formats: 0: cflowd, 1:pcap, 2:ASCII CSV, 3:MySQL, 4:wire, 5:PGSQL, 6:SQLite3
flow-tools RD version 0.67: built by eabarca@loadgen140 on Mon Aug 16 17:42:20 CEST 2010


./flow-export-v7 -l -f 3 -u [DB URI] < [FILE]
- Transactions in AUTO mode.
  TIME       # Records  KB Read KB Written
01:02:22        0       17695   42916592
01:02:34        100000  18207   42917168
01:02:46        200000  18419   42917608
```

DATASET AND TESTBED

DATASET DESCRIPTION

NAME:

Sample Netflow network data from netflow-tools FTP site.

DESCRIPTIVE ABSTRACT:

Dataset retrieved from Abilene network by ATLA in 2003-05-28.
ATLA is a Cisco Gigabit Switch Router (GSR) in Abilene network.
Abilene network is the old name for Internet2 USA Research network.

DATA ACQUISITION/COLLECTION SUMMARY:

Router was configured with sampled netflow with a sample rate of 1/100.
The data is anonymized by zeroing the last 11 bits of the IP address.

SOURCE:

ftp://ftp.eng.oar.net/pub/flow-tools/sample-data/ATLA/2003-05-28/

ARCHIVAL AND ACCESS INFORMATION:

ftp://ftp.eng.oar.net/pub/flow-tools/sample-data/README

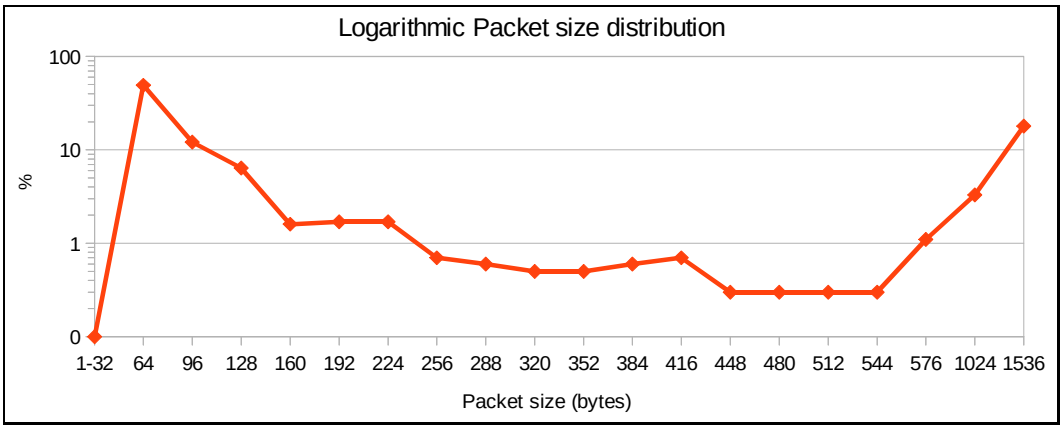
SPECIAL NOTES:

Abilene documentation: http://abilene.iu.edu
Internet2 website: http://www.internet2.edu
More and newest datasets under special arrangement available, instructions in: http://www.internet2.edu/observatory/archive/data-collections.html#netflow

SIZE:

- File size : 423 Mbytes, flow-tools compression enabled
- Total flows : Near 17 millions.
- Duration of data : 24 hours.
- Total Octets : Near 51.000 millions.
- Total Packets : Near 68 millions.
- Average flows / second : 196
- Average Kbits / second : 4706

PACKET SIZE DISTRIBUTION:



TESTBED DESCRIPTION

This work has been researched at our G-Lab[II] Routerlab in the Deutsche Telekom Laboratories, using Sun Microsystems Fire X4150 machines called *Loadgens* in collaboration with the Technische Universität Berlin (TU-Berlin).

Tests were made on similar Loadgens, internally numbered: #131, #137, #139 and #140. Development, initial research and tests were done in #131. As some tests showed slightly different results on different Loadgens, they were verified comparing results from #131/137/139. Finally, Loadgen number 140 was used to discover maximum performance reachable on final tests by establishing a faster I/O disk access.

As stated at the beginning, our research group prefer tools open to the community and all software used in this work has GNU/GPL license. Debian Lenny distribution together with MySQL, PostgreSQL and SQLite, flow-tools and monitoring tools like MRTG and DSTAT where constantly used. Any improvement in any of this software is publicy available.

HARDWARE CONFIGURATION:

All the four Loadgens have exactly the same hardware configuration:

Machine model:	Sun Microsystems Fire X4150
CPU:	2 Intel Xeon L5420 processors @ 2.50 GHz, having 4 cores each one.
RAM:	16 Gigabytes, though only 4 were used in our tests.
Storage system:	LSI SAS 1068E controller with four 174 Gigabytes SCSI hard disks

The only significant difference between Loadgens #131/137/139 and Loadgen #140 is that the first three of them were configured as Sun's RAID 1E[III][8] and #140 was configured as RAID 0 (Mirror mode) increasing I/O transfers up to the maximum before performing final results, here shown.

SOFTWARE CONFIGURATION:

A default standard Routerlab Debian Lenny 5.01 32bit Loadgen image with a 2.6.18-6 686 kernel, having being changed only to fix the stated storage system bug and upgraded to a 2.6.30.10 kernel was taken as a base system. Full system was running in 32 bit and only using 4 GB of RAM memory. The distribution maintained the Libc6 v.2.7 core libraries and the following packages were added to the base system:

- MySQL Server 5.0.51a-24+lenny2+spu1 with included MySQL client
- PostgreSQL Server 8.3.9 with included *psql* client
- SQLite 3 version 3.5.9
- dstat tool 0.6.7-1
- MRTG version 2.16.2-3
- flow-tools v. 0.68-12 (improved flow-export tool was based in 0.67)
- iostat sysstat version 8.1.2

Database server's configuration files were left as out-of-the-box with the following exceptions:

- Parameters changed as stated in this work, e.g. buffer synchronization.
- All databases pointing to the same Ext3 partition, different from the base system but still in

[II] Germany-wide research and experimental facility : http://www.german-lab.de

[III] RAID 1E uses 2-way mirroring on an arbitrary number of drives, leaving n/2 disk space and being tolerant to non-adjacent drives failing.

the same RAID-0 system.

Without suffering any changes, parameters related to performance tuning in MySQL were:

```
key_buffer              = 16M
max_allowed_packet      = 16M
thread_stack            = 128K
thread_cache_size       = 8
query_cache_limit       = 1M
query_cache_size        = 16M
```

And for the PostgreSQL server as follows:

```
shared_buffers = 16MB
checkpoint_segments = 3
```

SQLite3 had no configuration file at all in our experiments.

As for MRTG configuration the following SNMP MIBs were loaded:

```
/usr/share/snmp/mibs/UCD-SNMP-MIB.txt
/usr/share/snmp/mibs/TCP_MIB.txt
/usr/share/snmp/mibs/HOST-RESOURCES-MIB.txt
/usr/share/snmp/mibs/UCD-DISKIO-MIB.txt
```

And the following objects monitored (together with some parameters and routers.cgi script):

```
ssCpuRawUser.0&ssCpuRawSystem.0
ssCpuRawWait.0&ssCpuRawSystem.0
ssCpuRawIdle.0&ssCpuRawIdle.0
ssCpuRawUser.0&ssCpuRawUser.0+ssCpuRawSystem.0&ssCpuRawSystem.0+ssCpuRawNice.0&ssCpuRawNice.0
laLoadInt.1&laLoadInt.2
1.3.6.1.2.1.25.2.3.1.6.1&memCached.0
1.3.6.1.2.1.25.2.3.1.6.32&1.3.6.1.2.1.25.2.3.1.6.32 (host..hrStorageUsed)
iostat -kd /dev/sda3 (kB_read & kB_wrtn) (ext3 database storage partition)
```

TEST SETUP:

After several focused tests on kernel versions, hardware combinations, larger or experimental datasets and other specific situations, to compare database systems the basics of Experimental Design were followed. Specially the principles of *orthogonality* by means of altering only one factor on each test, *replication* principle by repeating tests until main variations appeared are identified or controlled and *blocking* principle as orthogonality provides us to.

There were 24 final insertion tests, all combinations of: 3 DBMS x 4 indexes x 2 field types, altering each time only one factor, creating this way:

- DBMS : MySQL / PostgreSQL / SQLite3
- Indexes : no index / index 0 / index A / index B
- Field types: generic (integer) / plaintext (specific)

For an easy interpretation of MRTG graphs, insertion tests had always the same execution order and was as in **Table 7**. This allow us to compare different graphs and interpret relation between User CPU, System CPU, I/O throughput, I/O wait time and System's Cache that will lead us to know the bottlenecks and test requirements and behaviour for performance improving.

Table 7: Insertion tests execution order

Ord.	Test	Ord.	Test	Ord.	Test
1	MySql-No Index-Integer	9	PgSql-No Index-Integer	17	SQLite-No Index-Integer
2	MySql-Index 0-Integer	10	PgSql-Index 0-Integer	18	SQLite-Index 0-Integer
3	MySql-No Index-Plaintext	11	PgSql-No Index-Plaintext	19	SQLite-No Index-Plaintext
4	MySql-Index 0-Plaintext	12	PgSql-Index 0-Plaintext	20	SQLite-Index 0-Plaintext
5	MySql-Index A-Integer	13	PgSql-Index A-Integer	21	SQLite-Index A-Integer
6	MySql-Index B-Integer	14	PgSql-Index B-Integer	22	SQLite-Index B-Integer
7	MySql-Index A-Plaintext	15	PgSql-Index A-Plaintext	23	SQLite-Index A-Plaintext
8	MySql-Index B-Plaintext	16	PgSql-Index B-Plaintext	24	SQLite-Index B-Plaintext

Tests were automated by the use of argument-based bash scripts to run all tests from scratch in a batch job, storing each test data in its own database, saving performance statistics on a results file. By fetching those statistic files and joining them into an spreadsheet we create the graphs showing the test behaviour. Many other special tests were executed to compare kernel versions, loaded storage system modules, different Loadgens, RAM memory available to the system, available free disk space, etc... to find a good test environment.

Two main scripts called *mult-insert.sh* and *mult-query.sh* run all the standard performance tests. The file *mult-insert.sh* calls the script *test-export.sh* with several parameters like database type (MySql/PostgreSQL/SQLite), index usage or not, type of index (none/0/A/B), type of field (generic/specific), transaction control, source data files used. Those parameters at the end correspond to a file name portion, having different files for each kind of task and joining all them together we construct the needed database structure. Those files contain specific commands to delete and re-create the database structure, indexes and fields. The *test-export.sh* file joins all them, empties all caches and buffers and executes the improved *flow-export* tool with the specified parameters in a pro-database-system base.

On the other side, the *mult-query.sh* script takes the part of querying data, it declares the SQL queries and the equivalent *flow-report* report executing timed queries on every configuration possible, saving each query result, time spent and user/kernel CPU used on a file. As our data will be written-once-read-many, queries are executed twice, the first one emptying buffers and caches, while the second one don't, this way we can compare the difference between the first time some data is used, and the followings. This is an important point because the second scenario will be the most common. At the same time MRTG with configured statistics (CPU, RAM, I/O) are being gathered for bottleneck monitoring purposes.

All tests have been run several times, contrasting results. Most graphs are showing mean values of three executions of the same test, all tests showed always the same behaviour and even peak differences were statistically not significant.

As stated before, we have a WORM (Write Once, Read Many) scenario, thus making the cache topic really important, and for this reason we will take care about cache behaviour in query tests. There were 48 final querying tests: all the previous combinations, plus another variant; with and without flushing caches after the previous query. This way we have statistics for the 24 scenarios, knowing the impact of the cache if we plan to query several times the same data, or mainly once.

Though not every variable in our tests was pretended to be controlled to make them 100% reliable as emptying the full database partition or running tests in different order, system's cache and buffers were cleared before each test by means of the new tunable option in kernels >= 2.6.16 that is

/proc/sys/vm/drop_caches. By means of running the following command: `sync; echo 3 > /proc/sys/vm/drop_caches` we order the system to sync the filesystem buffers and free the page cache, directory entries and inodes, obtaining thus the desired effect.

RESULTS

STORING DATA:

In **Figure 7** are shown the results from the most simple test, that will allow us to explain this kind of graphs. In this graph we can see the performance to insert the near seventeen million records, plotting a dot every 100.000 records (flows). This method will allow us to check scalability as more data is received and predict growth rate. On the Y-axis is represented the time in seconds to do the operation, while in the X-axis is shown the number of records (flows) written, from the first one to the last one. As easy seen, time spent to write records is linear along all the test, with minor spikes and clearly showing MySQL as being twice faster as the other database systems. Reason for that can be, for SQLite as not being a high-performance designed system and the usage of automatic field size and type detection and in case of PostgreSQL the need to store more data than MySQL as it does not support unsigned fields as explained before, requiring more I/O as demonstrated below.

In contrast, same test run with *Plaintext* field type showed slightly worse numbers, all databases required about two seconds more to write the data. This is caused by the amount of data processed and written, in e.g. MySQL with Integer field writes to disk 4.9 Mbytes/100K flows while in the *Plaintext* version writes 10.2 Mb/100K flows, SQLite writes 9.4 Mb/100K for the Integer field and 18 Mb/100K for the *Plaintext* version. There is a curious effect, PostgreSQL writes 51 Mb/100K on the Integer test and only 41 in the other and still is slower, this behaviour can be because its engine is not fast enough in this operation. This test demonstrates for all database systems using Integer fields performing better than Plaintext fields for insertion when not using indexes.

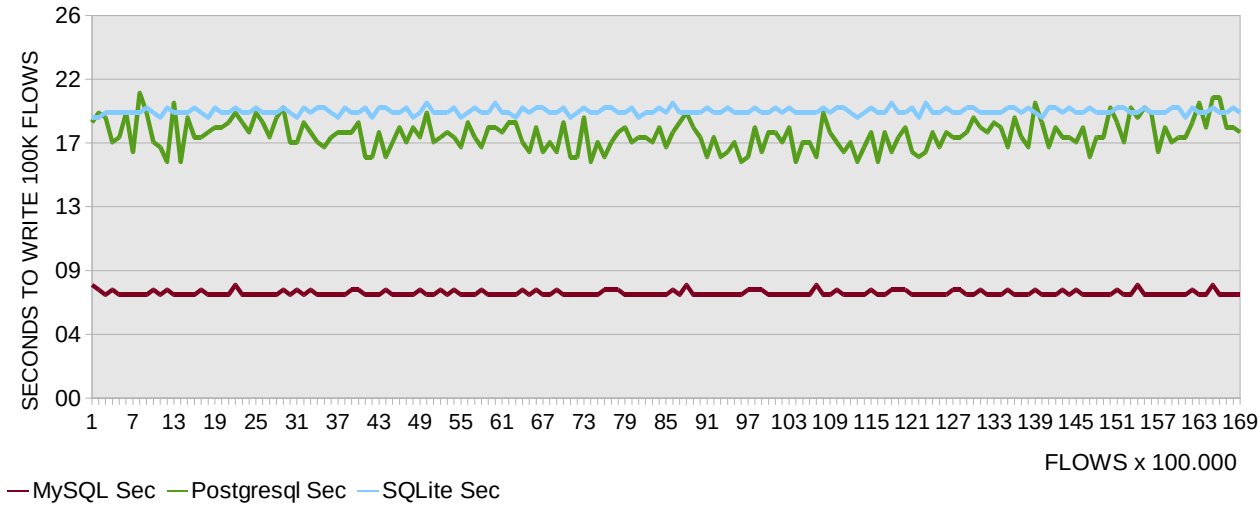


Figure 7: All DBMS, using Integer fields, no indexes, transactions: Auto

In **Figure 8** are shown all databases as before, with time to write in the Y axis (right) and a new variable on the Y axis (left): Megabytes written to the disk also every 100.000 records, those lines have a small right-pointing arrow to be recognized. In this experiment we are using Index 0, this is the reason for the big amount of data written to the disk, not only the flow data, it also counts index storage and modification. Here we can see bigger differences between database systems as this tests is more I/O demanding. MySQL is still the best solution as seems to write the lowest quantity of data and indexes, helping to do it quicker. We can appreciate that even if it is writing more data to the disk at the end, time spent to write records is constant, this means probably that flow-export tool is not fast enough to provide more data and it is the bottleneck in this experiment.

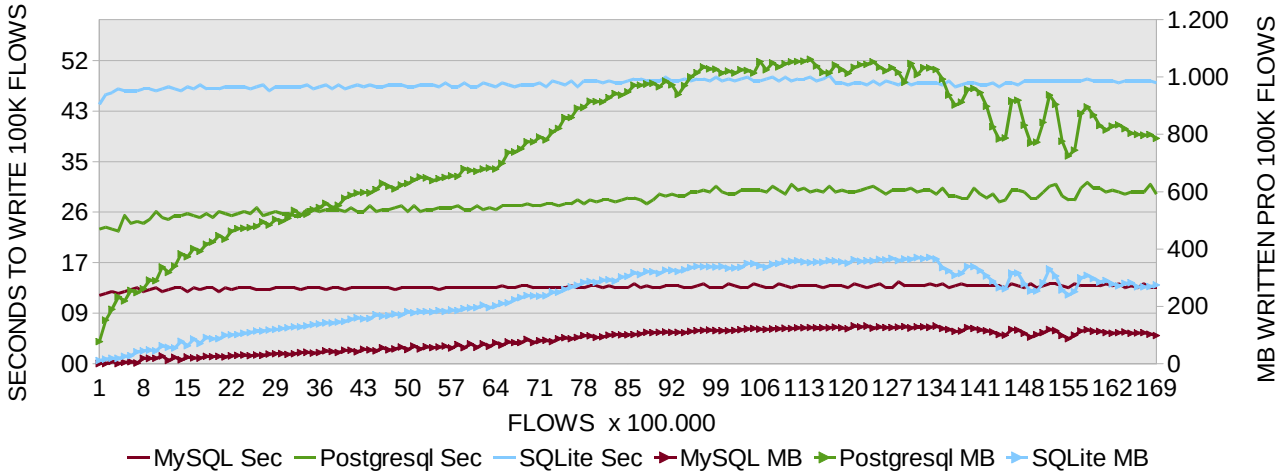


Figure 8: All DBMS, Integer fields, Index 0, transactions: Auto

In the case of PostgreSQL is also constant, but requires about twice time to do the same task as MySQL, if we pay attention to the curves of both PostgreSQL lines we can see some relationship between them, as disk I/O fluctuates time does also. SQLite in this test does not performs very well, even writing to the disk only twice as MySQL it requires about three times more than this one to do the same task. Probably as SQLite is not pretended to deal with this amount of data the indexing engine is not fast enough and creates a bottleneck which partially we can solve by using SQL transactions and is explained in the following **Figure** 9**.** At the end of the graph we can appreciate a strange behaviour in all databases with oscillating times and written data, this behaviour is explained later and called *data diversity effect*, as this is a data pattern effect that will appear in all graphs.

Figure 9 performs the same test as **Figure 8** with the exception of a single factor: transactions are forced to manual mode and set to 30.000 flows in size each transaction. This optimization made SQLite to be faster than PosgreSQL by only requiring around 21 seconds in contrast as the previously 47 seconds shown. If we compare blue arrow lines on both graphs we can appreciate the same amount of data written to disk, this is what makes us think about the indexing engine or internal operations as being the bottleneck on this indexing test. Other database systems were affected only slightly by using transactions. Other tests made with different number of flows pro transaction showed specific database small speed improvements, usually with values between 20.000-40.000.

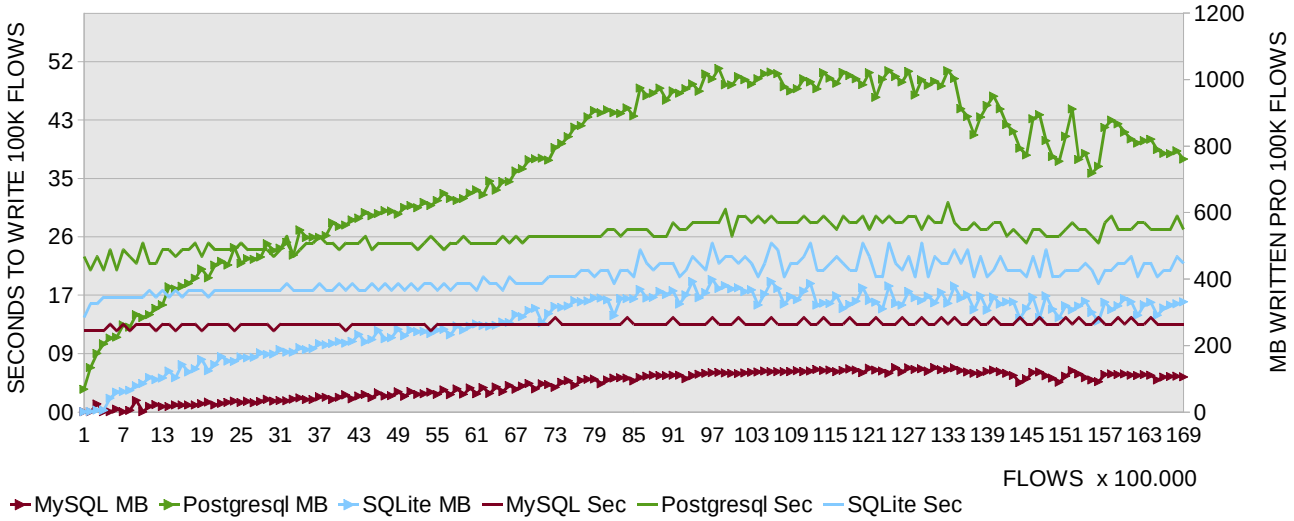


Figure 9: All DBMS, Integer fields, Index 0, transactions: forced to 30.000

Previous **Figure 8** and following **Graph 10** will allow us to compare between previous Index 0 and Index A. Should be noticed that both Y axis scales in this graph are narrower as this test is faster to execute though line shapes are similar. Even more, though both indexes have the same fields to index, Index 0 is composed only by single fields whereas Index A has half of those fields combined as described in the *Resolution* section. Even if amount of data to process is the same in both tests, the indexing of more fields has more overhead that the size of those fields, this is the reason for this test to be faster in all insertion cases. Depending on database index size the difference will be proportional, showing PostgreSQL a reduction around 19% on required time and data written. In facts of transactions, same principles apply to SQLite, using them will imply a good speed improvement, though number of flows pro transaction should be optimized again. Election of Index 0 or Index A will depend mostly on query performance. All databases perform in the same manner as seen by line shapes but in case those results are similar, Index A performs a bit better as expected.

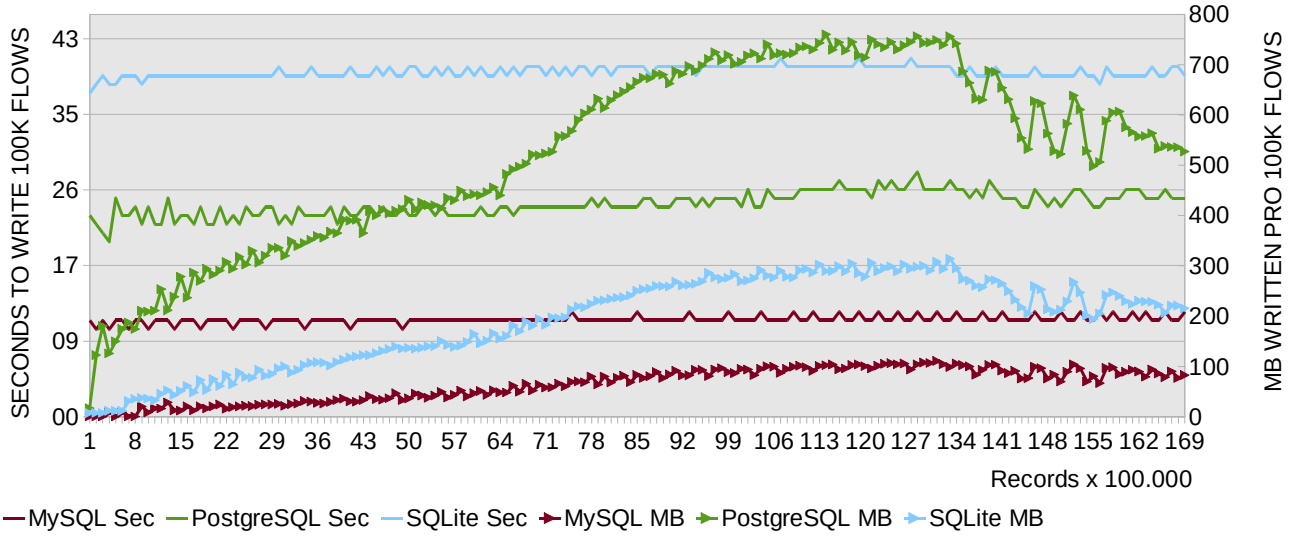


Figure 10: All DBMS, Integer fields, Index A, transactions: Auto

In **Figure 11** we can see performance of Index B, easily seen is that this one is the most resource demanding test. The reason is number of indexed fields and complexity of them. First, this Index B has two more fields to index: Src_IP_Port and Dst_IP_Port, both combined in a single one. And second, perhaps the complexity of combined records, in e.g. while in Index A [Src AS + Src IP Addr] field can create combinations of 65.536 (ASN) x 4.294.967.296 (Ipv4 Int32) but both numbers are often related to each other, in Index B [Src IP Addr + Src IP Port] results in the same amount of possible combinations but not so often correlated between them creating thus more different data. As the last one will require to write and maintain more data on disk this results in a higher I/O activity greatly effecting on seconds to complete operation.

Below we can appreciate how Y axis (right) is nearly three times than Index A I/O activity and Y axis (left) seconds is a bit higher. Worst results are given by the database system requiring more I/O, that is as always PostgreSQL. While the other database systems only increased I/O by two times factor, being it low, PostgreSQL did it by three scaling up to 2.300 Mb/100K flows. While in Index A PostgreSQL was faster than SQLite in this test twisted positions and together with the transaction optimization for SQLite it can advice us to better use SQLite than PostgreSQL in situations with several indexed fields.

Even with this increment in I/O usage, MySQL response to this test was really good, as it only performs low I/O it does not supposed a problem to it and time to complete operation was the same in

both tests. Only at the end when I/O requirements get to the maximum required about 17 seconds pro 100K flows in comparation with SQLite and PostgreSQL that both required about 52 seconds.

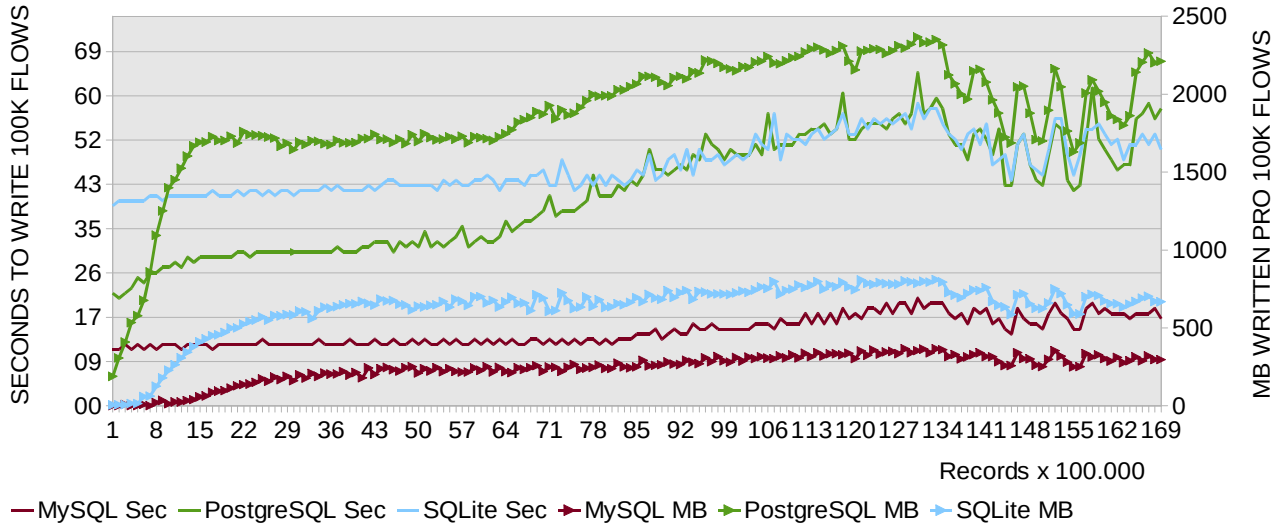


Figure 11: All DBMS, Integer fields, Index B, transactions: Auto

Figure **12** is the same previous test but with Plaintext fields. All database systems required more time and I/O to process the data. This demonstrates Integer fields as being better than the Plaintext version and last one should be only used when data conversion in data querying implies a big problem to deal with.

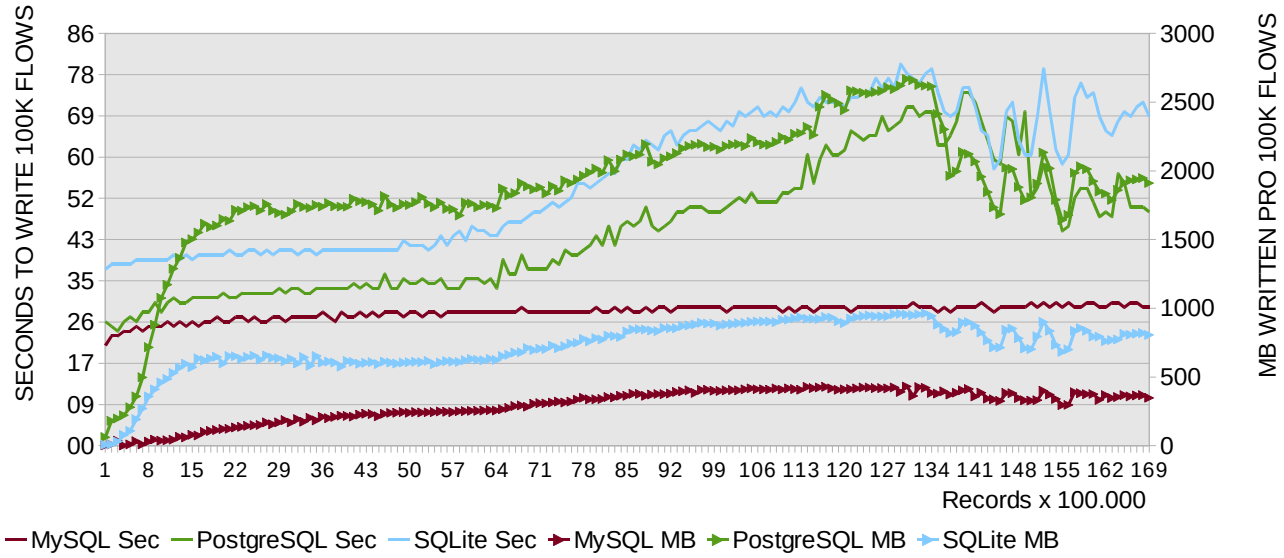


Figure 12: All DBMS, Plaintext fields, Index B, transactions: Auto

Data diversity effect and system scalability with indexes can be seen below in **Figure 13.** There is no need to do this test without indexes as it was already shown in **Figure 7** that line shape is linear and constant and should work without problems. Data generated for this experiment was from the *flow-gen* tool as explained on the *Resolution* section. This tool generates flows with field contents being numbers sequentially increased by 1. This creates weird flows with almost no relation between them at all, allowing us to find the worst-case situation and guessing system scalability.

This experiment was done with four times more data than previous tests, it has 66.2 million records. As our real data had only 17 millions, the same data was repeated by joining it four times,

having the common oscillating behaviour at the end. Through real data is not completely real for using this trick, the most important thing in this graph is the shape of the generated data. Generated data writes to disk about six times referred to real data, requiring twice the time to do it. At the beginning as data has no relation between them generates more I/O while creating indexes until it arrives to a point found in record 27 millions. The reason for that number can be that possible combinations stored in indexes begin to be similar/duplicated and thus 'related' allowing the indexing engine to optimize indexes or combine similar fields requiring then, less I/O transfers. Test demonstrates the importance of data diversity and that the MySQL database system is able to deal with probably any kind of traffic pattern and traffic size even using indexes.

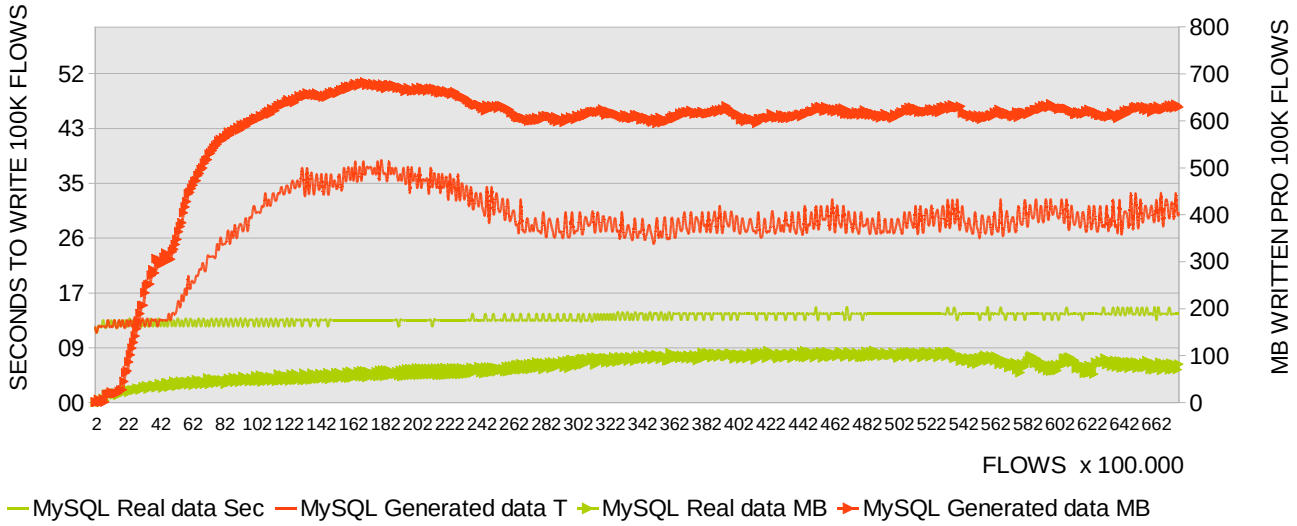


Figure 13: MySQL, Integer fields, Index 0, transactions: Auto

BOTTLENECKS

Specific test bottlenecks were found depending on test requirements. They were found thanks to, among others, MRTG.

In the case without indexes, bottleneck was the *flow-export* tool not being able to export data faster, though some optimizations can be done to the changes we did.

In overall, if we use indexes the problem is the no-multiprocessor support on the database system to manage them, I/O performance can be also a problem.

When querying data with indexes I/O performance is the bottleneck, in case without indexes and complex queries the no multiprocessor support will be the bottleneck.

SQLite executed extremely slow when using indexes, much more than the other DBMS, I think this behaviour occurs by the way its indexing engine is implemented.

It can be seen in **Figures** 14 to 17 some of those bottlenecks. In insertion graphs we can notice three things related to tests performance: time to finish each test marked by the width of each green column (specially in the cached memory graph). User CPU usage and behaviour of our flow-export tool measured by the height of the green column or CPU I/O Wait time depending on graph, and in the same manner but with the blue line, CPU used by system processes like database server. All four graphs are from the same set of experiments and are correlated, showing behaviour and conditions of the different tests. **Table 8** shows some test execution details, explained below, for those graphs as an example.

Table 8: Test execution details

CASE	TEST	BEGIN	END	Real duration	User CPU Time	System CPU Time
-	MySql (all)	17:10	00:27	07:17:00	-	-
-	PgSql (all)	00:27	11:20	10:53:00	-	-
-	SQLite (all)	11:20	03:21	16:01:00	-	-
A	MySql-no index-Integer	17:10	17:30	00:20:00	00:02:39	00:01:19
B	MySql-index A-Plaintext	21:15	23:08	01:53:00	00:03:45	00:03:01
C	PgSql-index B-Integer	06:05	08:03	01:58:00	00:03:24	00:03:55
D	SQLite-index 0-Integer	12:12	14:46	02:34:00	01:01:08	01:32:10

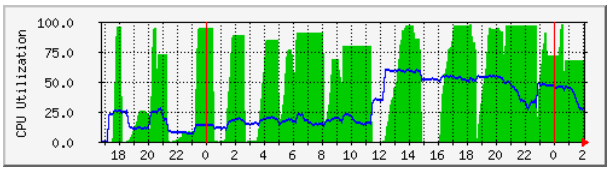


Figure 14: CPU: I/O RawWait(green) - System processes(blue)

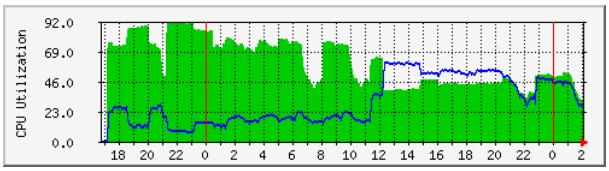


Figure 15: CPU usage processes: User(green) - System(blue)

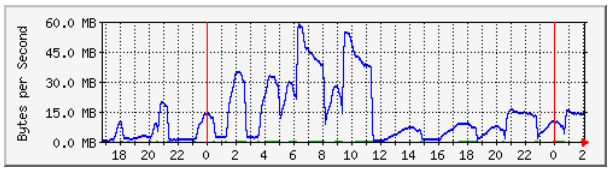


Figure 16: /mnt/databases Read(green) - Write(blue)

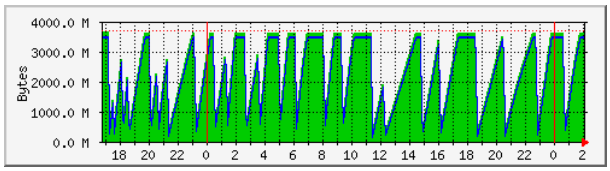


Figure 17: Memory Total/cached

CASE A

This test is the fastest one. As it only lasts about twenty minutes is very difficult to appreciate it in the graphs. As it is a no-index test, not as much data as others is written to disk, no I/O CPU wait time neither cached memory is required and it can be seen on the proportion between real duration (20 minutes) and User/System CPU (about 4 minutes) in this test and the proportion on the other tests.

CASE B

In this scenario the bottleneck is on the flow-export/client DB side; system CPU, CPU I/O wait and disk I/O is very low while User CPU is near the maximum. The whole system is waiting for the User CPU processes at all times. This can be caused for some data conversion and/or manipulation as this test needs to send more data as it is stored as plaintext format.

CASE C

Here is clearly seen how indexes and double size fields affect to PostgreSQL, it requires writing to disk about five times more data than other tests and system is waiting for this action to finish as wider and higher green columns for CPU I/O Wait shows.

CASE D

As SQLite is not a client/server database system it does not have a user process and a server process, instead everything runs as a user process and all System CPU spent time is not directly involved in internal database operations. Graphs show a very high System CPU time while only a medium-level User CPU time. This is probably caused because system is waiting for some unknown operation or high and long CPU I/O Wait time to finish and at the same time user-space processes are waiting the system

to finish internal operations. This unknown operation can be memory-related operations, it can be usual as SQLite is not designed for managing huge quantities of data.

Finally, to have a reference in **Table 9** are written some basic statistics about database sizes and best time to perform the 100,000 flows insertion:

Table 9: Database sizes and time to complete the insertion

Database	Field Type	Index	Size	Time to store
PostgreSQL	Integer	none	2.3 Gb	00:52
	Integer	Index-0	5.9 Gb	01:17
	Integer	Index-A	5.0 Gb	01:09
	Integer	Index-B	5.5 Gb	01:58
	Plaintext	none	2.3 Gb	00:57
	Plaintext	Index-0	5.9 Gb	01:23
	Plaintext	Index-A	5.0 Gb	01:09
	Plaintext	Index-B	5.5 Gb	01:58
MySQL	Integer	none	0.9 Gb	00:20
	Integer	Index-0	2.3 Gb	00:36
	Integer	Index-A	1.2 Gb	00:32
	Integer	Index-B	2.1 Gb	00:42
	Plaintext	none	1.8 Gb	00:25
	Plaintext	Index-0	3.8 Gb	01:30
	Plaintext	Index-A	3.0 Gb	00:32
	Plaintext	Index-B	3.6 Gb	01:19
SQLite	Integer	none	1.4 Gb	00:52
	Integer	Index-0	3.3 Gb	02:34
	Integer	Index-A	2.9 Gb	01:56
	Integer	Index-B	3.2 Gb	02:15
	Plaintext	none	2.8 Gb	01:15
	Plaintext	Index-0	5.9 Gb	02:32
	Plaintext	Index-A	5.4 Gb	01:56
	Plaintext	Index-B	5.7 Gb	02:15

QUERYING DATA:

In the next page, in **Figures 20 to 22** is shown an overview result of all tests: two different queries executed over all twenty four designed database combinations; executing them twice: cleaning and without cleaning system's buffers/cache. Flow-tools structure does not have any option to compare with the new database options, its results are repeated in each experiment for easy reading. Graphs at the left side are from Query A, at right from Query B. First row contains queries in databases using only integer fields while the second row were created with plaintext/specific fields. Each graph is showing results for all database systems, being the first bar of each colour for the first query execution, and the second one for the same query but without cleaning buffers.

System's cache helps improving time to execute by lowering it up to the half mainly in two situations: using MySql and indexes on the first query without being significant the type of fields, and for SQLite with the second query in all cases. Cache also helps every other tests by reducing the amount of time but only slightly.

Field type integer performs only slightly better on Query B but around three/four times faster in Query A than plaintext fields for nearly all database systems.

Too many factors can be compared on those graphs, but as we are searching for the best solution we can easily discard some of those combinations just looking graphs. SQLite is proved to be the slowest in all cases and with a great difference, we can discard it. Also there are no significant differences between Index A and Index B and most of the time Index 0 performs equally or better. No-Index versions seem to perform better than the remaining Index 0 version but we will keep this factor to take a closer look.

After simplifying those graphs we reduce complexity to **Figures 18 and 19** where it can be verified again that our indexing is not worth at all in any case. Focusing on Query A and No-Index bars the performance between all three systems is nearly the same, providing MySQL and PostgreSQL more flexibility and powerful language to retrieve what we need. In case of Query B results are not so good as both database systems require much more time to perform it.

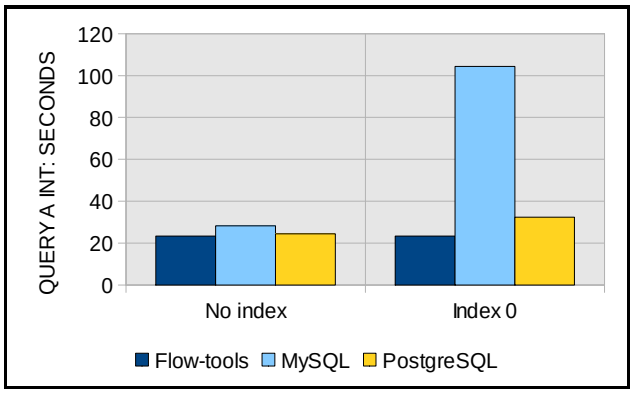


Figure 18: Query A, Integer fields

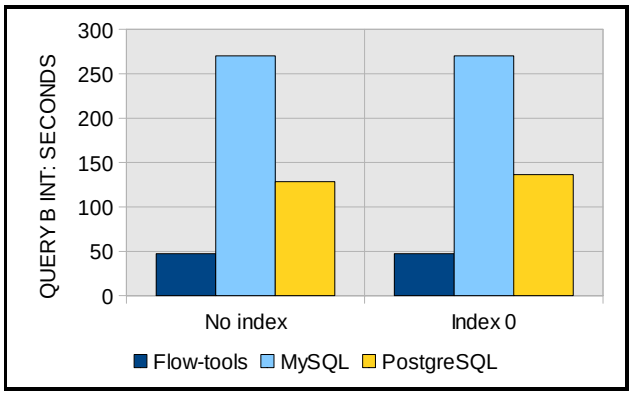


Figure 19: Query B, Integer fields

After these results some optimizations were done to the database systems demonstrating that better results can be achieved. For MySQL increasing *key_buffer* parameter and *query_cache_size* did not make any effect, but for PostgreSQL increasing available and cached memory reduced time to perform Query B-no index from 120 seconds to an acceptable amount of 70 seconds.

QUERY A **QUERY B**

INTEGER FIELDS

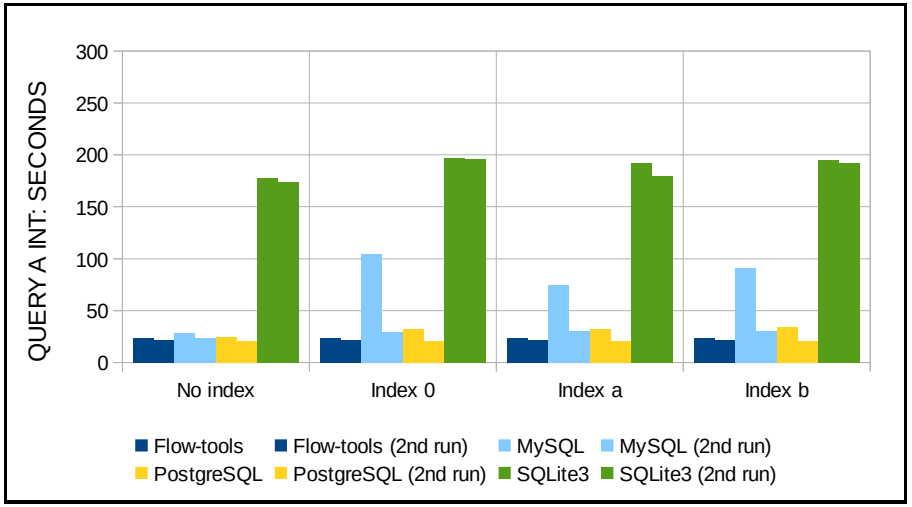


Figure 20: Query A, Integer fields, 1st and 2nd execution

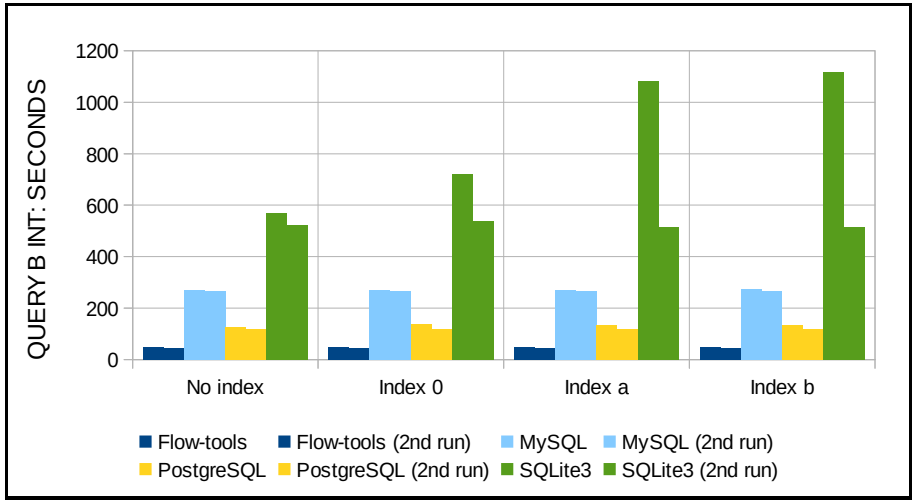


Figure 21: Query B, Integer fields, 1st and 2nd execution

PLAINTEXT FIELDS

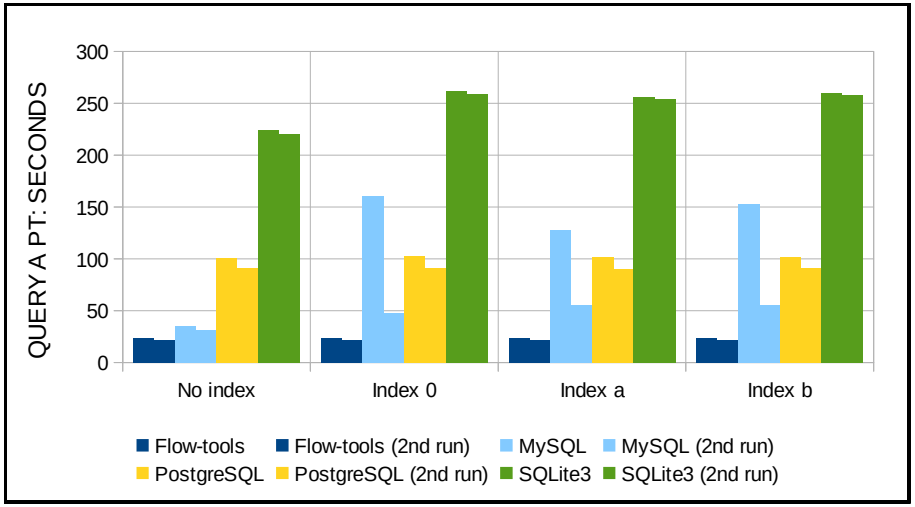


Figure 22: Query A, Plaintext fields, 1st and 2nd execution

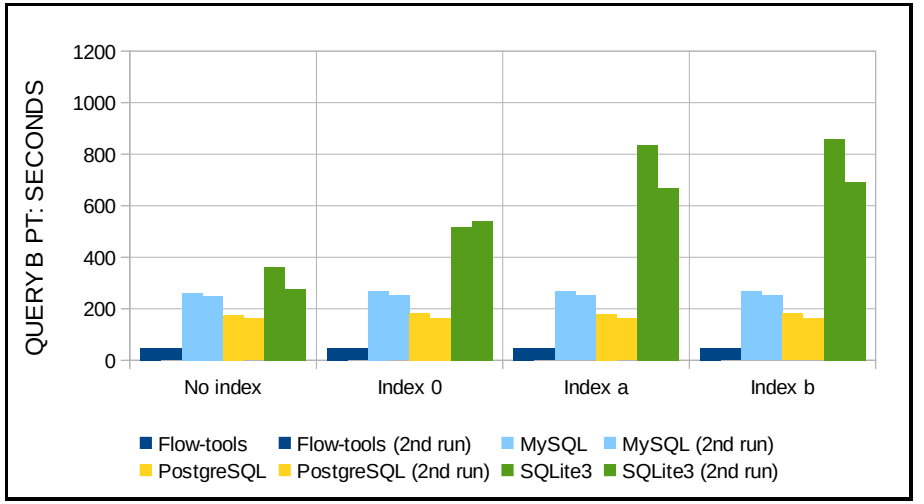


Figure 23: Query B, Plaintext fields, 1st and 2nd execution

CONCLUSIONS

In this work we have compared three database systems for the specific task of storing and accessing huge amounts of Netflow protocol data stored in databases. This comparation is not only to know wether it is possible or not to do it with databases, it is also to know which one of the three databases is the best for this specific scenario.

After many tests PostgreSQL showed that it will require a much faster and larger storage system than others, but in contrast complex queries will be executed two times faster than MySQL and a bit slower to *netflow-tools.* Simple queries are executed in PostgreSQL and MySQL as fast as *netflow-tools*. We should remember that MySQL only requires half storage space than PostgreSQL, it would be a good option in case we can not afford a large capacity storage system or one enough fast but we can wait some time for queries to be finished. At the same time MySQL is the fastest for data insertion and will be the best option again in situations where time-to-store is more important than time-to-query.

The use of transactions on insertion demonstrated that it greatly increases performance when using indexes and that only helps slightly when not using them. Database server optimizations have been left for future work and will allow databases to be nearly as fast as flow-tools, specially in PostgreSQL. All tests clearly show great improvements when using integer fields for time stamps and IP Addresses and the uselessness of using and creating indexes in real-time, leaving the creation of them if really needed to a later time in batch mode.

Database systems can perform similar to flow-tools in some queries providing a more powerful data query language but in other queries still do not perform so good. More research should be done to know the edge between simple-complex queries and take the right decision. Other table structures can also help on this by using database normalization and external tables to store Autonomous System extra data, port names, network aliases or DNS names.

In addition, future work can be directed towards improving our best solution with small improvements like bulk insertions with prepared SQL queries or big improvements like database compression, partitioning or the use of a cluster depending on storage or processor needs, but also trying different database systems for example some based on hierarchical data.

BIBLIOGRAPHY

[1] Pervasive Technology Labs (Indiana University): Netflow - What is it, and why do we hate it?
http://paintsquirrel.ucs.indiana.edu/pdf/netflow_hawaii.pdf

[2] Caligare Netflow history and documentation
http://netflow.caligare.com

[3] MySQL documentation
http://dev.mysql.com/doc/refman/5.0/en/

[4] PostgreSQL documentation
http://www.postgresql.org/docs/8.3/static/index.html

[5] SQLite documentation
http://www.sqlite.org/docs.html

[6] MySQL using Integer field type for IPv4 addresses
http://bafford.com/2009/03/09/mysql-performance-benefits-of-storing-integer-ip-addresses/

[7] Flow-tools web and *man* documentation
http://www.splintered.net/sw/flow-tools/

[8] Raid 1E disk system configuration
http://articles.techrepublic.com.com/5100-10878_11-6181460.html
http://en.wikipedia.org/wiki/Non-standard_RAID_levels#RAID_1E