DFRWS
DIGITAL FORENSIC RESEARCH CONFERENCE

A Scalable File Based Data Store For Forensic Analysis

By

Flavio Cruz, Andreas Moser and Michael Cohen

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2015 EU

Dublin, Ireland (Mar 23rd- 26th)

Digital Investigation 12 (2015) S90–S101

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

A scalable file based data store for forensic analysis

Flavio Cruz [a, *], Andreas Moser [b], Michael Cohen [b]

[a] *Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA*
[b] *Google Inc., Brandschenkestrasse 110, Zurich, Switzerland*

Keywords:
Distributed database
Incident response
Sqlite
Evidence analysis
Distributed computing

A B S T R A C T

In the field of remote forensics, the GRR Response Rig has been used to access and store data from thousands of enterprise machines. Handling large numbers of machines requires efficient and scalable storage mechanisms that allow concurrent data operations and efficient data access, independent of the size of the stored data and the number of machines in the network. We studied the available GRR storage mechanisms and found them lacking in both speed and scalability. In this paper, we propose a new distributed data store that partitions data into database files that can be accessed independently so that distributed forensic analysis can be done in a scalable fashion. We also show how to use the NSRL software reference database in our scalable data store to avoid wasting resources when collecting harmless files from enterprise machines.

Introduction

Digital Forensics necessarily deals with the storage, manipulation and exchange of large quantities of data, from disk images, memory images, to logical objects such as files, and analysis results (Garfinkel, 2010). In addition, practitioners do not only need to store large quantities of data, but they also need to be able to analyze it and ensure it can be easily exchanged.

Traditionally, proprietary storage formats such as the Eye Witness Format (EWF) have been developed to store evidence in structured containers (Guidance Software, 2014). Other proposals facilitate the free interchange of data, one example is DFXML which stores digital forensic information within an XML schema (Garfinkel, 2012).

The Advanced Forensic Format 4 (AFF4) was initially proposed as an interchange format for digital evidence (Cohen et al., 2009). The AFF4 proposal is essentially an object data store – objects are defined with appropriate behaviors and these are stored in the evidence file. The original AFF4 paper describes a data-at-rest file format centered around the Zip archive format and a number of objects with predefined behaviors (such a Containers, Streams etc). These objects are instantiated through a central *Resolver* which abstracts file storage details from the application.

The GRR Rapid Response (GRR) framework is a live forensic and incident response framework constructed using the AFF4 technology (Cohen et al., 2011). Rather than operating on static evidence files, the Resolver in GRR is implemented as an abstraction to a NoSQL data store. The application then uses the Resolver to permanently store AFF4 objects inside a NoSQL data store, while the rest of the application only deals with high level objects. NoSQL technologies are becoming increasingly popular in forensic analysis (Wen et al., 2013) since they offer more flexibility and scalability than relational databases (Parker et al., 2013).

The initial implementation of GRR was based around the proprietary BigTable technology (Chang et al., 2008) and demonstrates impressive scalability in remote response of very large numbers of machines. In the open source version of GRR that has since been released, the framework supports a number of interchangeable data store backends. By

* Corresponding author.
E-mail address: flaviocruz@gmail.com (F. Cruz).

http://dx.doi.org/10.1016/j.diin.2015.01.016

default, GRR uses a backend based on MongoDB (MongoDB, 2014). Other options include for example a MySQL (MySQL, 2014) backend. The scalability of the GRR system heavily depends on the performance of the data store technology, so choosing the underlying technology is extremely important.

In this paper we present a new data store backend that can be used as a storage layer for the AFF4 Resolver. We analyze the access patterns of AFF4 objects focusing specifically on the way that the GRR system utilizes the AFF4 space. By tailoring the data storage to the specific use case presented by GRR and AFF4, we implement a data storage layer that significantly improves the overall scalability of the GRR system in general.

This paper is organized as follows: First, we present the AFF4 object model and specifically examine how the GRR system utilizes the AFF4 abstraction. By analyzing the specific access pattern we propose a novel implementation of a NoSQL data store engine based on the SQLite database technology. We then evaluate the new data store in comparison to previous data stores. Finally, we utilize the new data store to perform a typical forensic analysis step – collect all the executable files on a Windows system which are not already known by the NSRL software reference database (NSRL, 2014b). The use of NSRL and other hash deduplication techniques has been demonstrated in the past to dramatically increase the efficiency of evidence collection and analysis, particularly for remote forensic applications (Rowe, 2012; Fisher, 2001; Watkins et al., 2009).

The AFF4 object model

The Advanced Forensic Format 4 (AFF4) was initially proposed as an interchange format for digital evidence that stores forensic data in object abstractions. All AFF4 Objects have a type, which specifies their behavior (e.g. An object of type AFF4Stream can be used to present an abstract stream interface), and a number of data attributes that contain additional information about the object (Cohen et al., 2009).

Every AFF4 object is identified by a Universal Resource Name (URN) which specifies an object uniquely within the AFF4 namespace. A URN is globally unique within the AFF4 universe and all access to AFF4 objects occurs via the *AFF4 Resolver* – a central logical factory for AFF4 objects. One can open, create and store AFF4 objects through the resolver, without consideration to their actual persistent serialization.

An important property of the AFF4 design is that the AFF4 namespace universe is assumed to be incomplete at any specific time. For example, when one obtains an AFF4 volume containing a number of AFF4 objects, it does not imply that we know the complete subset of the AFF4 universe. For example, an AFF4 object may refer to other AFF4 objects which are not necessary stored in that specific volume (i.e., there may be unresolved external references). This property allows merging different AFF4 volumes containing overlapping parts of the AFF4 namespace. Similarly, it does not make sense to directly enumerate any parts of the AFF4 namespace (since any specific implementation can not know the complete space). All AFF4 objects are related via semantic relations and therefore the AFF4 subsystem does not directly enumerate names, but must follow existing semantic links.

The following example illustrates this important point. Consider the logical collection of files on one machine's filesystem. The container aff4:/C.12345/fs/os/c:/Windows refers to the Windows directory of that filesystem. If we want to list the files contained within the Windows directory, we can not simply query the AFF4 subsystem directly to enumerate all URNs (e.g. with a wild card of aff4:/C.12345/fs/os/c:/Windows/*.*). Instead, we must explicitly store references to all children inside the AFF4-Volume object aff4:/C.12345/fs/os/c:/Windows itself, which are then used to retrieve the children of the directory.

The overall effect is that the data store must only support access to AFF4 URNs by exact name, rather than provide enumeration strategies. For the use of AFF4 in the GRR application this means that the application itself must maintain internal indexes to support object enumeration in cases where this is needed. For these reasons, modern key-value store NoSQL databases are a particularly good fit for serving the needs of the AFF4 data model (Grolinger et al., 2013).

The GRR rapid response framework

The GRR Rapid Response (GRR) Framework is a modern incident response and remote forensic tool designed to perform live forensics on a large number of systems. The GRR framework is outlined in Fig. 1. Although the details of the system are specified elsewhere (Cohen et al., 2011), the most pertinent point of this architecture is that GRR is constructed over the AFF4 subsystem. In practice, this means that all data stored in the GRR data store consists of serialized AFF4 objects. The AFF4 Resolver which allows

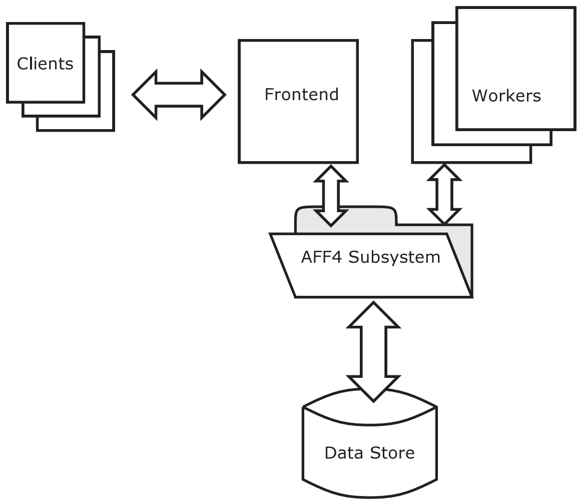

Fig. 1. The GRR Architecture. Clients use the HTTP protocol to exchange messages with the Frontend servers. Frontend servers in turn communicate with the AFF4 subsystem to queue messages in the data store. Workers communicate with the AFF4 subsystem in order to perform analysis tasks and schedule new operations on the clients. Note that all parts of the GRR framework interact with the AFF4 subsystem, which in turn abstracts access to the data store.

accessing those objects is implemented as a data store abstraction, which can be switched between a number of backend implementations.

Section 4 details all the requirements from a data store implementation but Fig. 1 already gives a clear indication that since the data store underlies all operations in GRR, it is critical to overall system performance and scalability.

Data store requirements

We summarize the functionality that a data store must implement in order to support an AFF4 Resolver:

- *Single object access*. Objects are only accessed by directly specifying their URN. The data store does not need to enumerate multiple objects at a time and all data store methods only operate at object level. This property simplifies the partitioning of data because operations never deal with multiple objects.
- *Support for both asynchronous and synchronous operations*. Synchronous operations will block until the data store returns the results, while asynchronous operations will be scheduled to be performed at some point in the future. Asynchronous operations follow an *eventual consistency* semantics and any subsequent data store operations may not reflect past asynchronous operations. Optionally, the data store provides a Flush() operation hat will wait until all asynchronous operations are fully completed.

Strictly speaking, asynchronous operations are not a prerequisite per se and could be replaced by just using their synchronous equivalent but experiments have shown that when processing large bulk inserts of data into the storage layer, asynchronous operations improve program concurrency and provide a huge performance advantage. Thus, it comes as no surprise that GRR makes heavy use of asynchronous operations. Nevertheless there are some operations of the GRR system which require synchronous operations to guarantee globally deterministic ordering.

- *Object locking*. The data store does not need to specifically support object locking, but must support atomic read-modify-write semantics. The AFF4 framework provides co-operative locking semantics built upon this feature.
- *Concurrency*. The data store must be concurrently accessible by multiple processes and threads.
- *Timestamped attributes*: The data store must maintain different versions of object attributes.

Since GRR is designed to scale horizontally, the data store APIs are specifically geared towards networked data stores. In such data stores, interactions occur over the network and, hence, may carry long latencies. The data store API allows for multiple operations to be specified over the same API call in order to amortize network latencies. So for example, it is possible to retrieve multiple AFF4 objects simultaneously, write multiple object simultaneously, etc.

GRR was initially released as an open source project with data store implementations based on two off the shelf, general purpose database technologies, namely MySQL (MySQL, 2014) and MongoDB (MongoDB, 2014). These database technologies are mature and perform very well for general purpose applications in multiple use cases. However, the design of databases is a compromise between performance and features. Since these database technologies are general, they necessarily perform sub-optimally for the GRR needs. For example, Section 2 describes the special optimization afforded by the AFF4 object model (namely that there is no need to maintain efficient enumeration capabilities in the data store). This optimization is not utilized by the general purpose data stores. Other possible backends for GRR would be distributed and clustered filesystems such as GlusterFS (GlusterFS, 2014) and pNFS (pNFS, 2014), however those lack timestamped data and the ability to perform complex queries.

It therefore makes sense that a custom data store, optimized to take advantage of the unique data access pattern of the AFF4 space will outperform the general purpose databases and filesystems.

SQLite data store

The data stores provided by GRR exhibit severe limitations:

1. *Horizontal scaling limitations*. As more workers are introduced to a GRR system, the capacity of each single worker is reduced due to contention at the data store.
2. *Storage limitations*. Since the existing data stores rely on a central data base server, increasing storage demands scaling the storage on a single server which is only possible to a certain extent.

We reasoned that many of the performance limitations noted above stem from the fact that the backend database tries to store all URNs in the same disk storage file, hence leading to file lock contentions. Our approach is to completely divide the AFF4 namespace into independent storage files. That is, we *shard* the AFF4 objects by their URNs across multiple containers.

This approach is shown in Fig. 2. Each URN is mapped into a separate SQLite file, and a specific handler is used to open it. The connection pool ensures that recently used

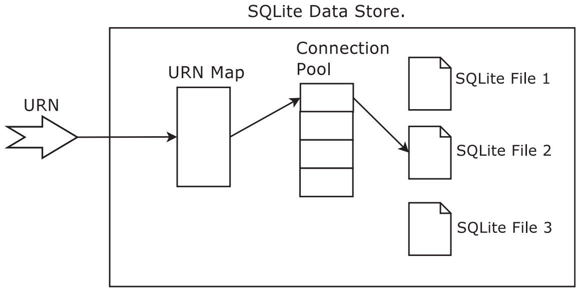

Fig. 2. An overview of the SQLite data store. For each URN presented to the API, the data store uses a *URN Map* to resolve it to a SQLite file name on the local file system. The data store then selects from a pool of connections a handler to manipulate that specific SQLite file.

files can be immediately reused without needing to reopen the file.

Organization

The SQLite data store is implemented as a directory of database files, where each file is a SQLite (SQLite, 2014b) database. We map the entire AFF4 namespace into this directory and each AFF4 object is mapped to exactly one file (even though one file can contain multiple objects). It is not possible for the attributes of an object to be split among multiple files. This simplifies operations since the data store always knows exactly where to apply a given operation.

Each database file contains two tables: tbl, which maps a triplet $(object, attribute, timestamp)$ to a $value$; and statistics, containing information about the database file. Fig. 3 shows the tables in each file. A value can either be a string, integer or a blob of data.

An index is added for the triplet $(object, attribute, timestamp)$ so that we can efficiently look up by object, by object and attribute, or by the full triplet. This index also acts as a primary key since there cannot be multiple values for the same triplet.

Mapping the AFF4 namespace

Objects such as aff4:/C.34a62f06/boot.ini need to be mapped to a unique file. The mapping algorithm uses a *mapping configuration* that is initially setup when the data store is created. A configuration is a list of regular expressions that maps objects to some specific path in the data store directory. The resulting path is always a substring of the object without the aff4:/prefix.

Let's consider the following configuration:

1. (?P<path>C.{1,16}?)/.*
2. (?P<path>hunts/[/̂]+).*
3. (?P<path>blobs/[/̂]+).*
4. (?P<path>[/̂]+).*

In order to assign a path to an object, the regular expressions are applied in order until one of them matches. We then retrieve the named group path that represents the path of the file.

For the object aff4:/C.4ecf7c33d24129c2/fs/os/boot.ini, the first regular expression will apply and return the path C.4ecf7c33d24129c2. Note that the first regular expression forces all objects related to a GRR client to be stored in the same file. The third regular expression maps URNs of the form blobs/ab29cf to a file ab29cf in the directory blobs.

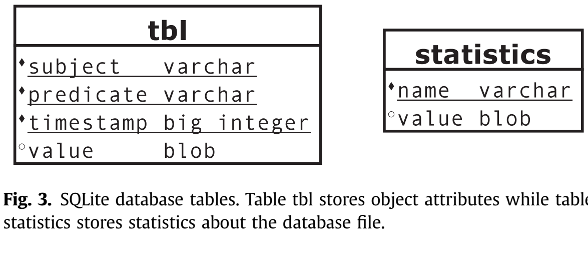

Fig. 3. SQLite database tables. Table tbl stores object attributes while table statistics stores statistics about the database file.

The sharding strategy is specified by the URN map. By choosing an appropriate strategy we must balance lock contention, file handle limitations and disk overheads. For example, in one extreme, we might map each URN to a separate database file. In this kind of configuration, files stay relatively small and operations on different URNs do not involve lock contention. However, in practice, there are limits on the total number of file handles a process may have so this will lead to a lot of opening and closing of database files. The overall disk usage will also be higher than necessary, since each SQLite file contains bookkeeping overheads.

At the opposite extreme, all URNs might be stored in the same file. This configuration requires less disk space but suffers from lock contention since all operations must lock the same table for updates.

A good mapping configuration creates a relatively high number of evenly sized files and groups related objects into the same file. This improves database locality, since the probability of doing an operation on an already opened file is high. It is also important that the number of queries per file stays relatively uniform since if one file is accessed too often compared to other files, throughput will again be reduced due to lock contention.

This is the reason we chose to implement the URN map as a configurable map of regular expressions. It is the application specific knowledge that is used to create a well tuned URN map: Frequently used parts of the AFF4 URN space are sharded into more files than less used subsets of the URN namespace.

SQL queries

The availability of the SQL language to perform queries on the database file makes it easy to implement the operations described in Section 4. Each operation that modifies the database creates a new SQLite transaction that is committed after all the commands are applied. For instance, a MultiSet operation may generate many INSERT commands – one per attribute that needs to be written – that are committed as a single transaction.

Concurrency and caching

SQLite supports multithreading and multiprocessing concurrency, which makes it possible to have different processes running queries on the same file at the same time (SQLite, 2014a).

Since there may be many SQLite files in the database directory, our implementation only keeps a limited number of them opened at any given point. There is a local cache of SQLite connections to speed up lookups but the cache is limited in size, closing off old connections to files that are no longer used.

Database maintenance

By default, SQLite database files will grow as new records are added. If a record is deleted however, the freed record is kept in the file to be reused for future inserts. This means that the database file is never reduced and with time, accesses will become slower with many interleaved INSERTs and DELETEs. To solve this, our data store runs the VACUUM (SQLite, 2014c) operation on files that have too many free records.

Typically, running VACUUM on a database might lead to a slow down as that file is compacted. However, since each SQLite file only contains a small subset of the URN space, the slowdown only affects operations on those objects while the rest of the system performs normally.

Distributed data store

While the SQLite data store has shown to perform well, it still shares one of the limitations with the existing data stores – the storage capacity is limited to a single machine. Even worse, the processes that use those database files must also run on the same machine since SQLite does not work well when files are stored on a remote filesystem like NFS (Sandberg et al., 1988).

As explained in Section 2, the AFF4 specification allows each AFF4 storage volume to only store a subset of the total AFF4 URN namespace. This allows for efficient sharding of the AFF4 namespace between multiple servers. By implementing a distributed data store that is able to run on clusters of computers (Buyya, 1999), we are able to greatly increase the storage capacity of our initial data store.

Architecture

An overview of the distributed data store architecture is shown in Fig. 4. Each process using the data store embeds a data store client library (E.g. the GRR frontend server, the GRR worker or the GRR GUI/CLI console). On the left, we have the *data store server group* that is composed of several *data store servers* and one *data store master*. The data store server group manages a database directory and each data store server manages its own shard. The data store master, itself a regular data store server, additionally performs special operations such as bootstrapping, data store server registration and data store maintenance.

The specification of a data store server group includes the addresses and ports of each server and an assignment of one of the servers with the data store master role. This specification is used by the data store master to create a *data store server configuration*. The configuration is stored in each data server database and contains not only the addresses and ports of all the servers, but the data store server shard map. On first execution, the data store master creates the data store server configuration and stores it in its own database. Every time a data store server starts, it registers with the data master and receives a data store server configuration that is also stored into the database.

Communication between data store servers and the data store master is done using HTTP. The data store master contacts the data store servers for maintenance tasks, while the data servers contact the data store masters for registration and for periodic pushes of data store server statistics. These statistics include server load, database size and number of files in the server.

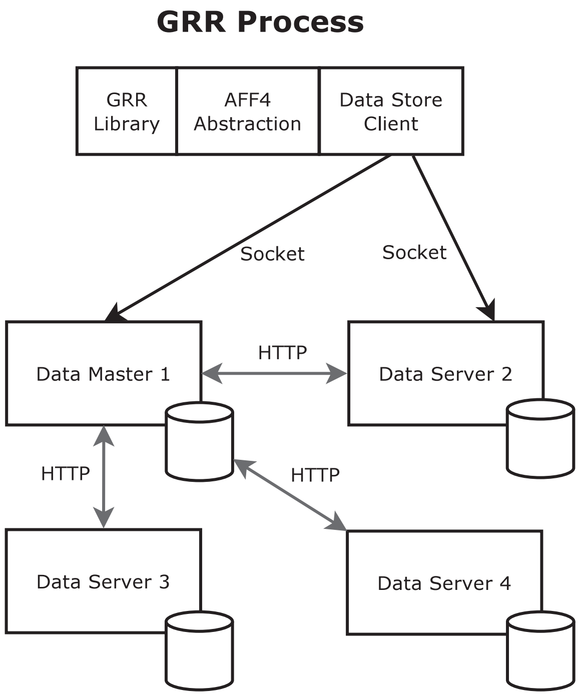

Fig. 4. Overview of the distributed data store. GRR processes use the data store client in order to make requests to the data store servers. Communication between data servers is done using HTTP, while streaming HTTP is used by the GRR processes in order to efficiently perform data store operations.

Mapping files to servers

When a client needs to use the distributed data store, it has to know which data store server to contact for a given object. That data store server, in turn, needs to map the object to a database file on its filesystem. The latter problem was already addressed in Section 5.2 through the use of the *URN Map*. The former problem (mapping URNs to data store servers) therefore, aims to uniformly shard database SQLite files among the available data store servers.

The data store server mapping therefore maps from SQLite filenames to the data store server which hosts this file. For this, we hash the SQLite filename path (obtained from the URN Map) to a 64 bit integer and then use the data store server mapping to locate the server hosting this file.

The data store server mapping contains, for each data store server, an interval in the range $[0, 2^{64}]$. The server intervals do not intersect and the union of the intervals is exactly $[0, 2^{64}]$. This mapping technique is known as *consistent hashing* (Karger et al., 1997) and allows us to smoothly add or remove data store servers. In Fig. 5 we present an example data store server mapping with four data store servers along with some objects. The range was uniformly distributed among the four data store servers and several objects were mapped into different servers by computing the hash value of their path. Since the distribution of the hash values is practically random and uniform, we expect the files to be distributed evenly among the servers.

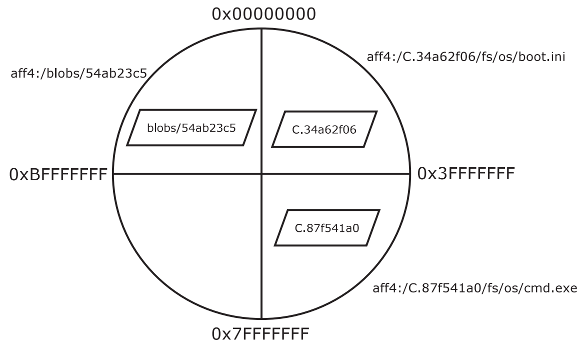

Fig. 5. Data store server mapping for four data store servers. The hash value distribution is uniformly distributed among the hash range, resulting in servers with an even number of files.

When a client starts using the data store, it randomly picks a data store server and asks for the data store server mapping. The mapping will subsequently be applied to all data store operations. The advantage of our design is that the data store master is not the sole owner of the mapping and is not responsible for mapping objects to data store servers. The clients only need the mapping to decide which data store server to use.

Fig. 6 shows how n URN finds its way from the client to the SQLite database file residing in the data store server. The mapping configuration is used in the Server Map and in the URN Map.

Operations

Every time a client needs to perform an operation on an object, it resolves the object to the data store server responsible for it. If the client does not have a communication channel already setup, it contacts the data store server and asks for a new *data store session*. Although the initial handshake is performed using the HTTP protocol, the data store server immediately changes to a faster streaming protocol.

A data store session is made of *N* pending *data store requests* and *M data store replies* from the server. A data store request is simply a data store operation presented in Section 4 and a data store reply contains the results of each data store request, including potential errors.

The use of streaming protocols for communication allows the client to write asynchronous data store requests into the server and then read the replies later. This improves the throughput of the system since there is no need to wait for the data store server response. For synchronous operations, the client will wait until it receives a reply for that operation.

In order to improve the concurrency of the system, the client maintains up to *C* communication channels to a single data store server, where $C > 0$. Since some communication channels may have pending requests, the client will select the channel with the least number of pending requests in order to get a reply faster. Fig. 7 shows a client with three communication channels to a data store server. The first channel has three operations and the data store server already replied to the two MultiSet requests. The second channel has two pending requests and the third channel has no pending requests, so the client decides to use this one for doing a MultiResolveRegex (a synchronous request that retrieves attributes of a given object, waiting for the results before returning).

Adding servers

After the data store is used for a while, many files will be created on each server, requiring the partitioning of more hardware. The problem with adding more servers is that the datastore server mapping will need to be adjusted as files are re-sharded onto the new server.

The distributed data store includes a *data manager* that allows changes to the data store server group. One of the use cases is adding new servers. Adding new servers works in two phases. First, a new server is added to the configuration but with an empty server range. This is followed by a *rebalance* phase, where datastore server map intervals are re-defined.

The re-balance operation is essentially a *two-phase commit* protocol (Mohan et al., 1986) and is orchestrated by the manager that communicates with the data store master. The data store master, then communicates with the whole data server group in order to synchronize the operations. A re-balance operation is thus done as follows:

1. **Configuration Phase**: The manager builds a new data store server configuration with the new ranges.
2. The manager sends the new configuration to the data store master and data master asks the data store servers to compute how much data needs to be moved around.

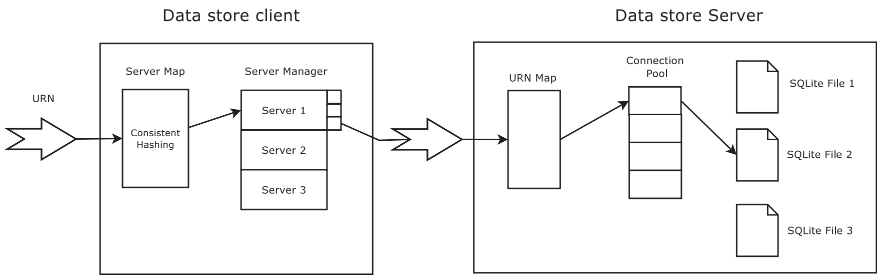

Fig. 6. From a URN to the database file. After mapping n URN to a specific data server, the data store client sends the URN to the server, where it will be mapped to an SQLite database file.

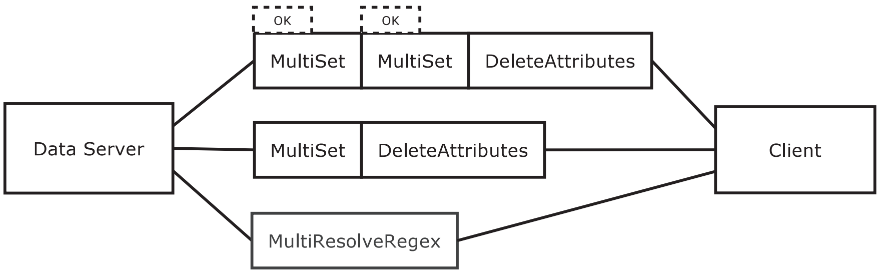

Fig. 7. Performing asynchronous operations on a data store server. Each data store client may create up to *N* communication channels in order to buffer many data store operations at once.

3. The data store servers will go through their database directory, map the files with the new configuration, and check if the file will stay in the same server.
4. **Commit-Request Phase**: The manager tells the user how much data will be moved. Once the user confirms, the manager will ask the data store master to force the group to copy the files to their eventual location. This phase is shown in Fig. 8.
5. Each data store server will receive a copy request and will then send the misplaced files to the new servers.
6. At this point, the new files will be stored in a temporary directory that represents a database transaction.
7. **Commit Phase**: Once all copy operations have been completed, the manager asks the data store master to perform the transaction.
8. Data store servers will move the files from the transaction directory to their correct location and those files that were sent are removed.
9. The operation completes.

If the operation fails during the Commit Phase, the data store is still able to recover from this error. The transaction can simply be resumed by forcing all the data store servers to move the remaining files from the transaction directory into the database directory, guaranteeing a clean data store state.

The use of consistent hashing allows us to move a limited number of files, since the server ranges will only change by a fraction whenever we add new servers.

Removing servers

Removing a data store server from the group can be done in two phases. We first change the shard ranges of the target server to be empty and then apply a re-balance operation with the new mapping. Once the files are fully moved, the data store server can then be safely removed from the server map altogether.

Experiments

We executed our benchmarks on an Intel(R) Xeon(R) CPU E5-1650 0 @ 3.20 GHz with 12 cores and 32 GB of RAM.

Micro-benchmarks

In this section, we present a set of micro-benchmarks that measure the raw performance of the data store when

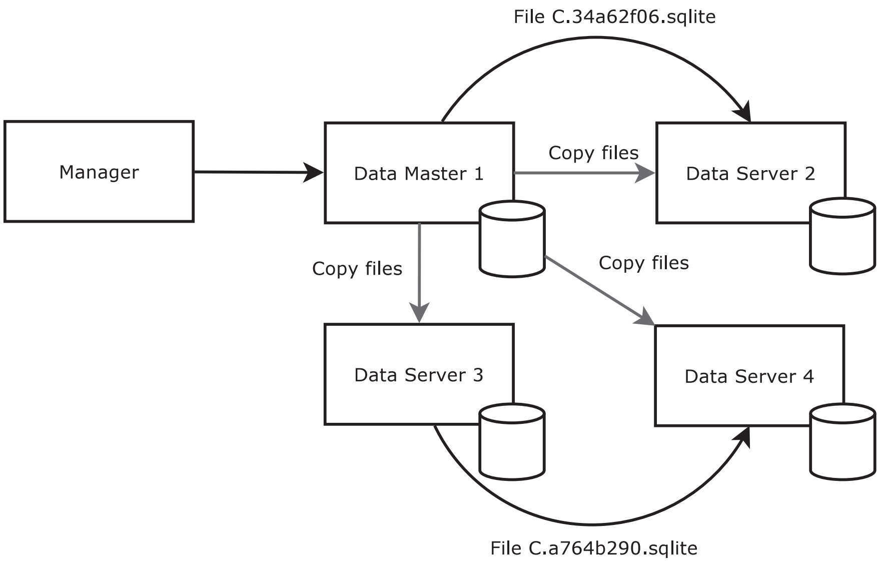

Fig. 8. Re-balancing the data store by copying files from server to server. The manager requests the data store master to coordinate the operation and the data store master makes sure all the data store servers send their files to their correct data store server.

performing a sequence of data store operations. We wrote a series of scripts that create a data store and initially fill it with a large number of AFF4 objects and corresponding attributes. Afterwards, they apply predefined sequences of read, update and delete operations. It is important to note that those scripts are single threaded and we therefore do not expect to see performance gains when using multiple data store servers.

We measured the time taken and total data store disk usage for each data store at each checkpoint the test suite. The data stores include MySQL, MongoDB, the basic SQLite data store and the distributed data store with up to four data store servers.

We now present three scenarios used in the benchmarks.

Many objects, each object having few attributes

Fig. 9 presents the results of a benchmark where the datastore is filled with 25,000 AFF4 objects, each having at most 3 attributes each. Each attribute has 3 versions and the attribute value is only 100 bytes long. The 25,000 objects are distributed within the namespace of 500 GRR clients (i.e. 500 SQLite shards). The *Values* line represents the total number of attribute values in the data store with the number of iterations.

After the database is filled with all the objects, we perform several data store operations. The *Values* line in the plot indicates if new values were added or removed. When the line is parallel to the horizontal axis, it means we are performing read operations of random objects. At the end of the benchmark, we remove all the objects from the data store and the number of values goes back to 0.

MongoDB takes a long time to complete (more than 700 s) and SQLite is the fastest, taking under 100 s to complete the full benchmark. The distributed data store, represented in the plots as *HTTP*, performs similarly to MySQL and shows very little differences in performance when using either one, two or four data store servers. This is reasonable since the micro-benchmarks are sequential (i.e. there is no concurrency).

In terms of size, MySQL has the smallest data store size (only 17 MB), while SQLite comes up in second place with 30 MB. The SQLite data store creates 500 shard files, which represent the 500 GRR clients. The distributed data store is not shown in the plots since it has the same size as the SQLite data store.

Few objects and many attributes

For the second benchmark, presented in Fig. 10, we start with a data store with only 100 objects and for half of them,

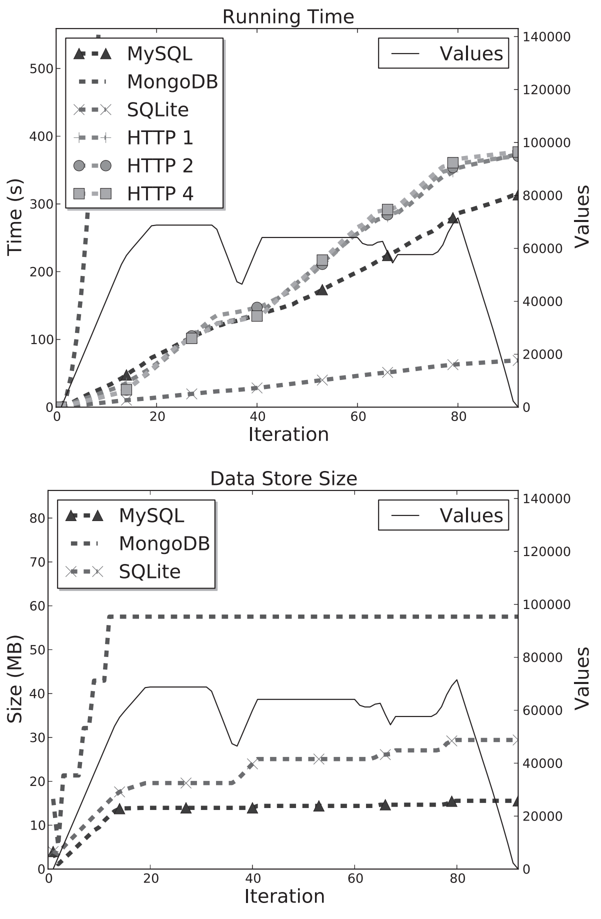

Fig. 9. Many objects with few attributes.

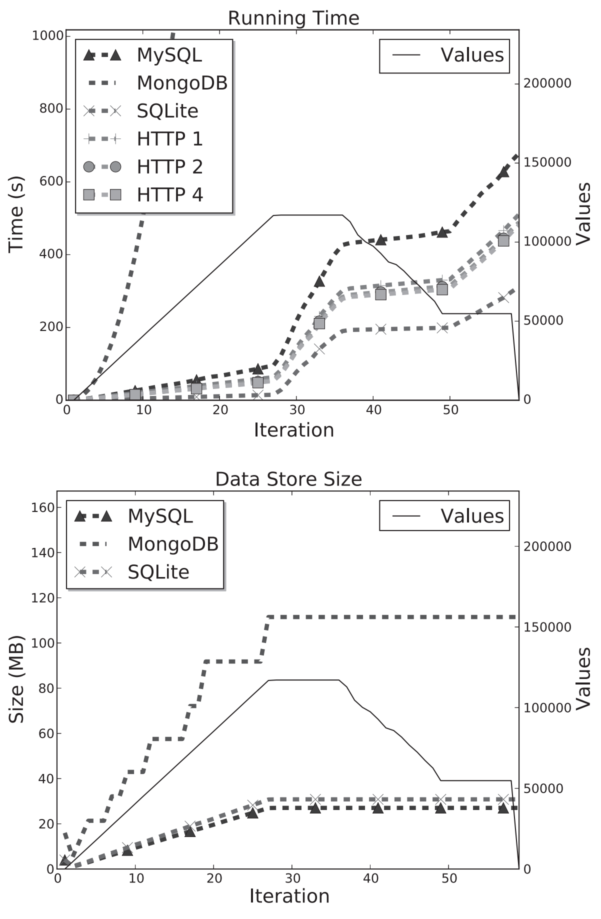

Fig. 10. Few objects with many attributes.

we add 1000 attributes. The 100 objects are distributed among 5 GRR clients (i.e. 5 database files).

Once again, SQLite is the fastest. However the gap between SQLite and MySQL has closed when compared with the previous micro-benchmark. Interestingly enough, the distributed data store is now faster than MySQL. The differences between them increase during the first read operation of the database (around iteration 30–40), indicating that MySQL performs slow reads once the database is filled up with many values.

In terms of size, both SQLite and MySQL have very similar data store sizes. Since SQLite is only using 5 files, there is little overhead wasted on many database files. However, as noted, the performance dropped slightly.

Many objects and many attributes

Finally, in Fig. 11, we present the last scenario, where we have 500 GRR clients with 50 AFF4 objects each. Each object has 50 attributes. This is the longest running micro-benchmark since the total number of attribute values is the highest.

The SQLite data store outperforms all other data stores in terms of run time, but still uses more storage than MySQL. We do not show the MongoDB data store in this plot since the MongoDB data store was unable to complete this micro-benchmark in a reasonable time. We also note that the distributed data store is slightly faster than MySQL.

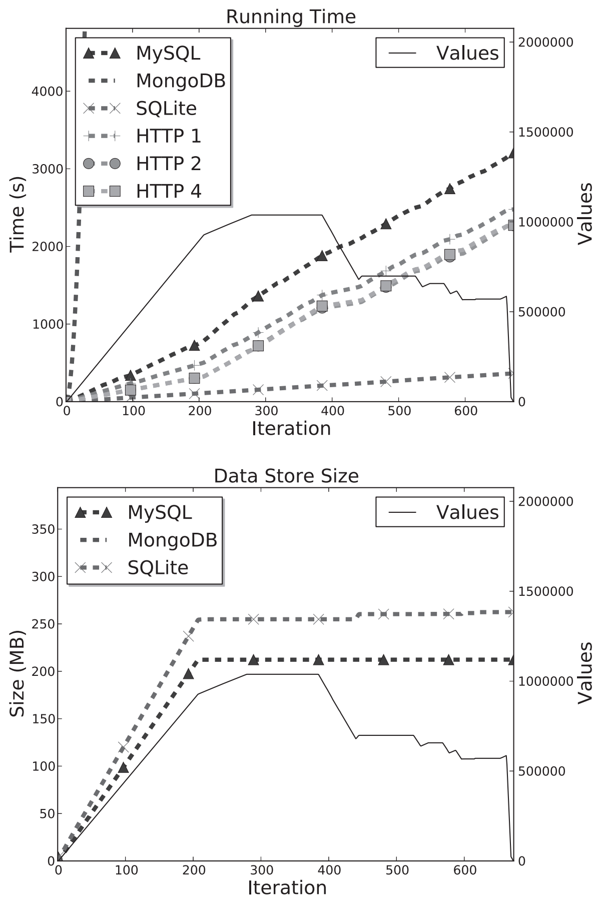

Fig. 11. Many objects with many attributes.

End to end benchmarks

The previous section examined how the data stores perform when running the micro-benchmarks. However, in practice, the overall performance of the system depends on other factors than pure data store performance. In order to get a more realistic feel for the scaling performance in typical system operation, we designed a set of end to end benchmarks where we run the GRR system with 100 GRR clients connected and then perform multiple *Flows* (Cohen et al., 2011) on the clients. All the GRR processes, including the GRR clients, are executed on the same machine. To better stress the data store, all the flows are started at the same time. We register the time elapsed between flow creation and flow completion.

GRR supports the use of multiple frontend servers, which communicate with the clients, and multiple workers, which process work stored in the data store by the frontend servers. Theoretically, increasing the number of frontend servers and workers should reduce the time needed to complete all the flows. However, if the system is not scalable, run time will only reduce slightly. For these benchmarks, we use three configurations:

1. A single frontend server with one worker
2. Two frontend servers and two workers
3. Four frontend servers and four workers.

Fig. 12 presents the results of the end to end benchmarks for the SQLite, MySQL and the distributed data store. We do not present the results for the MongoDB data store since those benchmarks just take too long to finish on that backend.

The SQLite data store (Fig. 12(a)) is the fastest data store and one of the most scalable. Most flows are completed under 100 s when using four frontend servers and four workers.

The MySQL data store (Fig. 12(b)) shows relatively good overall speed but poor scalability. While the SQLite data store is able to reduce the run time in almost half when using two frontend servers and two workers, the MySQL data store can only reduce the run time by 25% percent. The situation gets worse when using four frontend servers and four workers, since the run time barely improves upon the previous configuration. The poor scalability of MySQL may be attributed to the fact that the data store is using a single MySQL database, which all processes will need to synchronize at every datastore operation.

For the distributed data store, we measured the performance by varying the number of data store servers. We experimented with one, two and four data store servers and the results are summarized in Fig. 12(c–e). It can be seen clearly that there is a clear improvement of performance as we add more data store servers. When using only a single data store server, there is only one process handling all the requests from the different GRR processes. Even though the requests are distributed among several threads,

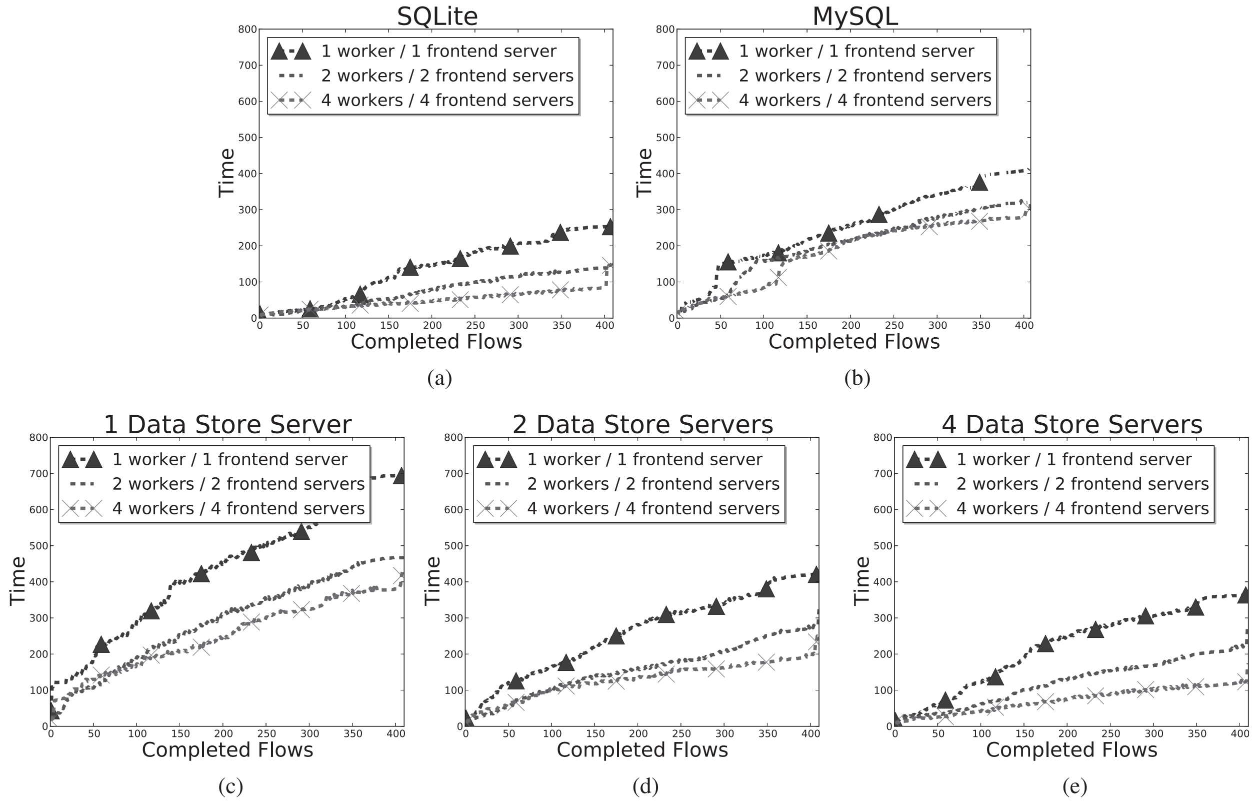

Fig. 12. End to end benchmarks using a different number of frontend servers and workers. As we increase the computing capacity, we expect the data store to scale and allow the run time of the system to go down almost linearly. In our experiments, only MySQL is not able to scale successfully.

a single data store server process will only have limited capacity when dealing with multiple clients. A big part of this problem can be attributed to the Global Interpreter Lock (Python, 2014) used by the Python language to synchronize multithreaded byte-code execution. However, these limitations start to disappear as we add more data store server processes. When using four data store servers, we see that the performance of the distributed data store scales very well and the overall run time starts to approach the SQLite data store and clearly outperforms MySQL in all configurations.

We also measured the size of the database of each distributed data store after running the end to end benchmarks. Table 1 presents the size and number of files found in each data store server directory. The results show that the files are evenly distributed across the data store servers.

Table 1
Size of the data store for each data store server after executing the end to end benchmarks.

Server	Size (MB)	# files
Server 1	26	58
Server 2	27	52
Server 3	29	53
Server 4	29	50

NSRL hash de-duplicated file collection

The National Software Reference Library (NSRL) contains metadata about known files present in known software packages. Each file entry of the database contains the cryptographic hash values of the file's content, file name, file size and the software package containing the file. The NSRL is commonly used in forensic analysis to exclude known files from further manual analysis (NSRL, 2014a).

The GRR system already de-duplicates files based on hashes when retrieving the files from GRR clients. So for example if the same file is present in different clients, GRR will recognize that the file hash exists in the data store and will not retrieve the file from the client – saving bandwidth, time and additional storage. By pre-populating the data store with NSRL hashes it is therefore possible to prevent GRR from downloading known NSRL files again into the data store, saving resources.

We created a new type of AFF4 object representing the NSRL file (This AFF4 object does not have data contents, only hashes and metadata). During the initial import stage, we write a single NSRL AFF4 object for each entry. The AFF4 objects are stored into the AFF4 namespace under the scheme aff4:/files/nsrl/sha1-hash. When GRR fetches a file from the client, it checks if an object exists at this URN, and decides if the file should be transferred.

Using our local SQLite data store, it took us five hours to import the complete dataset into a directory of 33 GB in

Table 2
Size of the data store for each data store server after importing the NSRL library.

Server	Size (MB)	# files
Server 1	8216	1001
Server 2	8739	1065
Server 3	8362	1020
Server 4	8286	1010

size. To split the dataset into many SQLite files, we defined a *URN Map* configuration that splits the NSRL hash objects using the first three characters of the SHA-1 hash value, so that all the entries that share the first three characters will be stored in the same SQLite database file. Table 2 shows the number of files and database size after importing the NSRL library into a distributed data store composed of four data store servers. Again, we notice that our sharding mechanism works very well, with an almost perfect partitioning of data across servers.

We also imported the NSRL library into MySQL and it took over 3 days, around fifteen times longer than it did using the SQLite data store. We consider this long import time and also the following slow data access prohibitively expensive for day to day use and, thus, we conclude that our novel data store approach makes utilizing data sets as big as the NSRL to deduplicate live file collection feasible in GRR for the first time.

We then applied our pre-populated data store to collect all executable files residing on a fresh Windows 7 SP1 machine. By manual analysis, we found 1605 executable files occupying a total of 479 MB of disk space. We intend to assess how much space and time we save by checking if a given file is already present in the NSRL library and thus does not need to be downloaded from the GRR client. We start a flow on the GRR system that collects all the files ending with .exe on the Windows client. The client will hash the files locally and send back the SHA-1 hash value that is checked against the imported NSRL library. If the file is found, we do not download the file's content from the client and do not store it in the data store since it is a well known file referenced by the NSRL library.

Table 3 presents the results of our collection experiment. Using the NSRL library, we note a 77% reduction in the number of files fetched from the client machine and a reduction in the network usage and CPU time that would be needed to transfer the skipped files. We argue that the use of our new data store allows us to import large data sets of useful forensic information that can be used to reduce the time and space it takes to store new data. Although we have used only a single client machine, if we had to do the same forensic analysis on multiple machines, the overall number of files fetched and stored would be reduced accordingly, resulting in even greater data de-duplication. Furthermore, due to our scalability results we showed early one, our data store can be used to perform concurrent collection of many machines, speeding up remote forensic analysis.

Table 3
Collecting executable files from a GRR client. When importing the NSRL library into the distributed data store, we avoid duplicating data by checking if the file is already referenced by the NSRL library.

Statistic	With NSRL	Without NSRL
Files found	1605	1605
Files skipped	1245	2
Files fetched	360	1603
Data store size	148 MB	314 MB
Client sent	117 MB	243 MB
Client received	5 MB	8 MB
Client time	293 s	400 s

Conclusions

We have presented a novel distributed data store for GRR, an incident response framework that uses the AFF4 object model to store and retrieve forensic data. This new implementation leverages specific properties of the access pattern exhibited by the GRR application to improve data processing performance and, by extension, the scalability of the entire GRR system. As another advantage over the existing data store implementations, our new approach also seamlessly shards the data to be stored among a number of data servers to further increase scalability.

In the presented experiments we have shown how this data store approach clearly outperforms the existing implementations. In the micro-benchmarks we have run, the SQLite data store is on average five times faster than the second fastest implementation (the one based on MySQL) and still more than twice as fast in the worst case. In addition, we have shown that the distributed data store we introduced shows the best scalability characteristics by obtaining a 2.9-fold speedup from using four GRR workers instead of one where the MySQL data store can only leverage a 1.38-fold performance increase. Finally, we have shown how this new data store is able to efficiently deal with big amounts of real world data by storing the complete NSRL hash set and utilizing it to perform data deduplication while downloading files from Windows machines in a live GRR test setting.

We believe that demands on storage technology are rising steadily due to the increasing number of computing devices holding more and more information. This is an important challenge that needs to be addressed by the forensics community and our data store design is a promising approach to tackling this challenge and allowing forensic practitioners to gather and analyze large amounts of forensic data.

References

Buyya R. High performance cluster computing: architectures and systems. Upper Saddle River, NJ, USA: Prentice Hall PTR; 1999.

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, et al. Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst (TOCS) 2008;26(2):4.

Cohen M, Bilby D, Caronni G. Distributed forensics and incident response in the enterprise. Digit Investig 2011;8:S101–10.

Cohen M, Garfinkel S, Schatz B. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. Digit Investig 2009;6(Suppl. (0)): S57–68. The Proceedings of the Ninth Annual {DFRWS} Conference.

Fisher GE. Computer forensics guidance. Management 2001.

Garfinkel SL. Digital forensics research: the next 10 years. Digit Investig 2010;7:S64–73.

Garfinkel S. Digital forensics xml and the dfxml toolset. Digit Investig 2012;8(3):161–74.

GlusterFS. Glusterfs distributed filesystem. 2014. http://www.gluster.org.

Grolinger K, Higashino WA, Tiwari A, Capretz MA. Data management in cloud environments: nosql and newsql data stores. J Cloud Comput Adv Syst Appl 2013;2(1):22.

Guidance Software I. Encase forensic. 2014. http://www.guidancesoftware.com/products/.

Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. STOC'97. New York, NY, USA: ACM; 1997. p. 654–63. URL, http://doi.acm.org/10.1145/258533.258660.

Mohan C, Lindsay B, Obermarck R. Transaction management in the R* distributed database management system. ACM Trans Database Syst 1986;11:378–96.

MongoDB. Mongodb, the no-sql database. 2014. http://www.mongodb.org/.

MySQL. Mysql database system. 2014. http://www.mysql.com.

NSRL. Data formats of the nsrl reference data set (rds) distribution. 2014. http://www.nsrl.nist.gov/documents/Data-Formats-of-the-NSRL-Reference-Data-Set-12.pdf.

NSRL. National software reference library. 2014. http://www.nsrl.nist.gov.

Parker Z, Poe S, Vrbsky SV. Comparing nosql mongodb to an sql db. In: Proceedings of the 51st ACM Southeast Conference. ACMSE'13. New York, NY, USA: ACM; 2013. 5:1–5:6.

pNFS. Parallel nfs/nfs v4.1. 2014. http://www.pnfs.com.

Python. Global interpreter lock. 2014. https://wiki.python.org/moin/GlobalInterpreterLock.

Rowe NC. Testing the national software reference library. Digit Investig 2012;9:S131–8.

Sandberg R, Golgberg D, Kleiman S, Walsh D, Lyon B. Innovations in internetworking. Norwood, MA, USA, Ch: Artech House, Inc.; 1988. p. 379–90. Design and Implementation of the Sun Network Filesystem.

SQLite. File locking and concurrency in sqlite version 3. 2014. http://www.sqlite.org/lockingv3.html.

SQLite. Sqlite website. 2014. http://sqlite.org.

SQLite. Vacuum command. 2014. http://sqlite.org/lang/_vacuum.html.

Watkins K, McWhorte M, Long J, Hill B. Teleporter: an analytically and forensically sound duplicate transfer system. Digit Investig 2009;6: S43–7.

Wen Y, Man X, Le K, Shi W. Forensics-as-a-service (faas): computer forensic workflow management and processing using cloud. In: Cloud computing 2013, the Fourth International Conference on Cloud Computing, GRIDs, and Virtualization; 2013. p. 208–14.