Storlet Engine for Executing Biomedical Processes within the Storage System

Simona Rabinovici-Cohen, Ealan Henis, John Marberg, and Kenneth Nagin

IBM Research – Haifa,
Mount Carmel, Haifa 31905, Israel
{simona,ealan,marberg,nagin}@il.ibm.com

Abstract. The increase in large biomedical data objects stored in long term archives that continuously need to be processed and analyzed requires new storage paradigms. We propose expanding the storage system from only storing biomedical data to directly producing value from the data by executing computational modules - storlets - close to where the data is stored. This paper describes the Storlet Engine, an engine to support computations in secure sandboxes within the storage system. We describe its architecture and security model as well as the programming model for storlets. We experimented with several data sets and storlets including de-identification storlet to de-identify sensitive medical records, image transformation storlet to transform images to sustainable formats, and various medical imaging analytics storlets to study pathology images. We also provide a performance study of the Storlet Engine prototype for OpenStack Swift object storage.

1 Introduction

Two trends are emerging in the context of storage for large biomedical data objects. The amount of biomedical data objects generated by various biomedical devices such as diagnostic imaging equipment, medical sensors, wearable devices and genomic sequencers, is increasingly growing both in the number of objects and in the size of each object. Additionally, these large data sets which may be stored in geographically dispersed archives over many years, need to be continuously maintained, processed and analyzed to reveal new insights. A second trend is the arising of new highly available, distributed object/blob stores that can serve lots of data efficiently, safely, and cheaply from anywhere over the Wide Area Network (WAN). Unlike traditional storage systems that have special purpose servers optimized for I/O throughput with low compute power, these new object stores are built from commoditized off-the-shelf hardware with powerful CPUs.

We propose to marriage these two trends and leverage the new storage system processing capabilities to execute computations. We define *Storlets* as dynamically uploadable computational modules running in a sandbox within the storage, and propose them for biomedical processes. While with traditional storage systems, analyzing biomedical data requires moving the data objects from

the storage system to the compute system, we propose to upload data-intensive biomedical storlets to the storage system and execute the processing close to the data residency.

For example in the pathology department, a tissue block is cut off into thin slices and mounted on glass slides. The thin slices may then be stained with different stains, e.g. HER2, PR, ER, and images are taken for the original and the stained slices. These images may be very large up to 200K over 200K pixels and can consume about 5-10 GB storage. Later on, business processes may be initiated to mine and analyze those large pathology images. Here are some examples:

- **ROI Extraction**. The pathologist wants to extract a region of interest (ROI) from several images taken from the same tissue block. In the traditional flow of this business process, all the many images of the tissue block are downloaded from the hospital storage infrastructure to the pathologist machine. He then applies on his machine an *Image Alignment* module to align the various images. Next, he selects a few images and apply *ROI Extraction* module to extract the relevant box from them. Afterwords, a *Format Transformation* module may also be applied to transform the data from its propriety format to a standardized one that can be rendered on the pathologist machine. With storlets technology, we can substantially improve the efficiency of this business process. The pathologist will trigger the *Image Alignment Storlet*, *ROI Extraction Storlet*, and *Format Transformation Storlet* to run within the storage infrastructure. Only the final result of the transformed ROI image will be downloaded to the pathologist machine.
- **Cell Detection**. In continuation to the previous flow, the oncologist now wants to analyze the ROI images created by the pathologist. The oncologist gets the ROI images and applies *Cell Detection* module to detect what are the cells, their shapes and count in the image. The *Cell Detection* module is based on heavy computer vision algorithms that require a lot of CPU, memory and even special hardware (GPU). The oncologist machine may be too weak to execute this module, and thus fails to produce the result. With storlets technology, the *Cell Detection Storlet* will run in the cluster of machines within the storage. Only the final result will be successfully displayed in the oncologist machine. Moreover, when the *Cell Detection* module is updated, e.g., due to new computer vision algorithms, we just need to upload the updated storlet to the storage infrastructure which is much less effort than downlaoding all the images to the oncologist machine again.
- **Cohort Identification**. A researcher would like to get a cohort of images with similar features from as multiple patients as possible. In the traditional business process, an hospital assistant would need to download all images from the storage infrastructure to a staging compute system. Then, the assistant would apply *Image Similarity* module to create a cohort, followed by a *De-identification* module to comply with HIPAA. Finally, the de-identified cohort is sent to the researcher. With storlets technology, the *Image Similarity Storlet* will run in the cluster of machines within the storage. Then, the

De-identification Storlet will de-identify the data within the storage and send to the researcher. This is more secure than the traditional method as it spares the need to move clear data to the staging system.

To summarize, the benefits of using storlets are:

1. **Reduce bandwidth** – reduce the number of bytes transfered over the WAN. Instead of moving the data over the WAN to the computational module, we move the computational module to the storage close to the data. Generally, the computational module has much smaller byte size than the data itself.
2. **Enhance security** – reduce exposure of sensitive data. Instead of moving data with Personally Identifiable Information (PII) outside its residence, perform the deidentification and anonymization in the storage and thereby lower the exposure of PII. This provides additional guard to the data and enables security enforcement at the storage level.
3. **Save costs** – consolidate generic functions that can be used by many applications while saving infrastructure at the client side. Instead of multiple applications writing similar code, storlets can consolidate and centralize generic logic with complex processing.
4. **Support compliance** – monitor and document the changes to the objects and improve provenance tracking. Instead of external tracking of the objects transformations over time, storlets can track provenance internally where the changes happen.

To enable the use of storlets, the storage system need to be augmented with a *Storlet Engine* that provides the capability to run storlets in a sandbox that insulates the storlet from the rest of the system and other storlets. The Storlet Engine expands the storage system's capability from only storing data to directly producing value from the data.

The concept of storlets was first introduced in our paper [1], where storlets were used to offload data-intensive computations to the Preservation DataStores (PDS). It was afterwards investigated in the VISION Cloud project[1]. Our main contribution is the introduction of storlets for biomedical processes and the definition of the Storlet Engine for OpenStack Object Storage (code-named Swift)[2]. We describe the architecture of the Storlet Engine as well as the programming model for storlets. The security model of the Storlet Engine supports storlets multi-tenancy and various types of sandboxes. We also provide a performance study of the Storlet Engine prototype.

The Storlet Engine was developed as part of PDS Cloud [2] that provides storage infrastructure for European Union ENSURE[3] and ForgetIT[4] projects. The Storlet Engine is used to process archived data sets in medical, financial, personal and organizational use cases. Various data sets and storlets were examined for the projects; some described in the above example business processes.

[1] http://www.visioncloud.eu
[2] https://wiki.openstack.org/wiki/Swift
[3] http://ensure-fp7.eu
[4] http://www.forgetit-project.eu

2 Storlet Engine Architecture

The Storlet Engine provides the storage system with a capability to upload storlets and run them in a sandbox close to the data. Figure 1 illustrates the Storlet Engine architecture that is generic with respect to the object storage used. It includes two main services:

- **Extension Service** – connects to the data-path of the storage system and evaluates whether a storlet should be triggered and in which storage node.
- **Execution Service** – deploys and executes storlets in a secure manner.

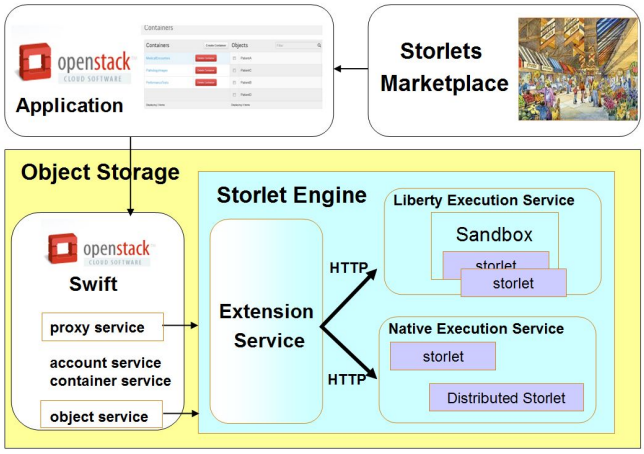

Fig. 1: Storlet Engine Architecture

The extension service is tied to the storage system at intercept points and identifies requests for storlets by examining the information fields (e.g., HTTP headers, HTTP query string) in the request and by evaluating predefined rules. The extension service then performs a RESTful call to one of the execution services that executes the storlet in a sandbox according to the client request and its credentials. The extension service needs to adapt the hooks into the object storage and may need for that either the source code or an open interface to internally connect to the storage system. We prototyped the Storlet Engine for OpenStack Swift Storage that is open source.

The Storlet Engine supports multiple execution services and multiple programming languages for storlets. It supports execution of Java storlets in an execution service based on IBM Websphere Liberty Profile product that is a web container with small footprint and startup time. We also support native execution service for storlets written in Python or other languages that are wrapped in a Docker[5] container, which is open source software to pack, ship and run an application as a lightweight Linux container. A storlet may call other storlets; even if the other storlet is in a different execution service.

[5] https://www.docker.io

The Storlet Engine can reside in the storage interface node (e.g. proxy node in Swift), and in the storage local node (e.g. object node in Swift). We implemented the ability to run storlets either in the interface proxy servers or in the local object servers to take advantages of each node underutilized resources. The performance study in section 4 shows that it's preferable to run data-intensive storlets in the local object node. Yet, for storlets that are compute intensive or access several objects in different object servers, it may be preferable to execute them in the interface proxy node.

A Storlets Maketplace can be used as a repository of storlets from different vendors. An application on top of the storage can mashup and use different storlets from the marketplace for creating its functionality. For example, an HMO can create in the Marketplace a deidentification storlet that complies with HIPAA. A PACS provider can create in the Marketplace an imaging analytics storlet. Then, another application can use the two storlets from different vendors to analyze deidentified medical images and correlate them with deidentified medical records.

2.1 Storlet Development

A storlet is a data object in the object store that includes executable code. The storlet code is based on the standard Java servlet technology; thus the storlet developer may use the extensive existing development tools for servlets. The storlet includes three aspects:

- **Lifecycle management** – includes operations such as storlet deployment, storlet configuration and initialization, processes and threads management, storlet execution, inter-storlet communication. Lifecycle management is provided by the Storlet Engine.
- **Business logic** – the actual functionality of the storlet. This is provided by the client or application that creates the storlet.
- **Services invocation** – calls performed by the storlet to the Storlet Engine for external functionality e.g. to access additional objects, call other storlets, perform integrity check.

2.2 Services Storlets

The Storlet Engine provides special services storlets that can be used either by external clients or called by other running storlets. One such service storlet is the Distributed Storlet. The Distributed Storlet is a compound service storlet that executes multiple storlets in a pre-defined flow. It is a compound storlet in the sense that it calls other storlets and consumes their results. It is a service storlet as it is provided by the Storlet Engine itself, and not by external developers. This storlet is intended for analytics processes where distributed data-intensive processing on multiple objects is required.

The input parameters to the Distributed Storlet include the *resources* data objects to process, the *split* storlets to activate on the resources and the *merge*

storlets that combine the results. We have used the Distributed Storlet to perform cells detection in pathology images. The Cell Detection storlet includes a complex time-consuming algorithm. Thus, the Distributed Storlet activates multiple Cell Detection storlets in parallel on each data object at its residence. Then, it waits for all split storlets' results and calls the merge storlet to combine the detected cells to one image.

Conceptually, the Distributed Storlet has similarity to MapReduce programs that transform lists of input data elements into lists of output data elements [3]. However, the Distributed Storlet is specialized for distributed processing within the storage system, sparing the data transfer to the compute nodes.

Another service storlet is the Sequence Storlet which is a compound service storlet that executes multiple storlets one after the other. The output of one storlet is the input to the next storlet in the sequence. The output of the final storlet is returned to the user. The Sequence Storlet is provided by the Storlet Engine as all services storlets.

Those services storlets are written in Python and run in the native execution service.

2.3 Rules Mechanism

The goal of the rules mechanism is to allow implicit storlet activation via predefined conditionals, thereby enabling automatic conditional activation of storlets. The storlets implicitly invoked by rules are in addition to the explicit storlet activation. Normally explicit storlet activation, if requested, takes priority and overrides implicit storlet activation. However, rules are also used to enforce access control related actions by marking the request with a specially marked flag. For such requests passing through the rules mechanism is mandatory. For example, mandatory de-identification storlet on sensitive data is enforced for requests having limited access credentials.

The rules mechanism is implemented via a rules handler that implements the rules logic (per stored rules), in combination with the user and system parameters. The rules handler is part of the extension service in each one of the Swift nodes that includes the Storlet Engine.

Typical inputs of the rules handler include request parameters, system metadata (e.g., content-type), user metadata (e.g., role, tenant), and the stored rules set. The output of the rules handler includes a full specification of the objects, parameters and REST request headers required for invoking a storlet.

Examples of executing storlets on the basis of predefined rules and various input parameters include: a) de-identification storlet (depending on the role attribute of the user in the request), b) content transformation for the entire container (depending on the content type of the object provided in the request).

3 Security Model

The Storlet Engine is an extension of the storage system. As such, it needs to adhere to the following security requirements.

- Storlets should execute in a protected environment where users (clients) do not over-reach their privileges. Specifically, access to data (e.g., Swift objects) from within a storlet should be allowed according to the user's permissions.
- In a multi-tenant storage environment, full isolation among tenants is a fundamental concept. This must be extended into the storlet's environment. Storlet requests coming from a given tenant should not be exposed to any aspect of requests coming from a different tenant (including data, state, etc.).
- Storlets originate from different sources. The code is not always fully verifiable for safety and consistency. The storlet engine must guard against malicious storlets, as well as intruding requests wanting to abuse the storlets.
- Multiple storlets serving multiple clients co-exist in the storage system. The storlet environment should support scalability. Storlets must not consume excessive resources (cpu, memory, etc.) that might degrade the performance of the system to an unacceptable level.

3.1 Sandboxes

When a storlet is deployed, it is placed in a "sandbox", that controls and limits the capabilities and actions of the code. A sandbox typically can block undesirable actions, such as direct access to cloud storage, or communication outside the local machine. It can also restrict access to the local filesystem and to local ports.

Currently two types of sandboxes are available, associating different levels of trust with different storlets:

- **Admin Sandbox:** the storlet is fully trusted, has no restrictions, and can perform all actions.
- **User Sandbox:** the storlet is less trusted, and therefore restricted from performing certain actions, such as writing to the filesystem except in designated work areas, reading from areas of the filesystem that are irrelevant to the storlet, making arbitrary network connections, or issuing sensitive system calls.

3.2 Implementing Multi-Tenancy and Sandboxes

We implement a lightweight sandbox mechanism, leveraging capabilities of the underlying operating system. It does not rely on unique properties of a given storlet execution environment, and can potentially be applied in multiple different storlet execution environments in the same system. The only assumption is that the operating system is of the Linux type.

In Liberty Profile, each web server executes in a separate single Linux process. Each running process has an effective UID (Linux user identifier) and GID (Linux group identifier), as well as a real UID and GID. Consequently, all the storlets deployed in a given web server have the same UID and GID.

In our implementation, a web server provides a single type of sandbox for all storlets that are deployed in that server. In addition, we restrict each server to be engaged on behalf of a single tenant. A given storlet can be deployed

independently in multiple servers, thus separate instances of the same storlet can be made available in each sandbox for each tenant.

Each tenant/sandbox pair is associated "permanently" with a unique UID and GID. In particular, different sandbox types of the same tenant have different UIDs/GIDs. All servers run as non-privileged (non-root) processes. The UIDs/GIDs are leveraged to support filesystem permissions, firewall policies, and tenant isolation. Each tenant/sandbox pair is also associated with a unique port number – on which the server listens for storlet requests.

An effect of the unique UID/GID approach is that a well behaved server and its storlets can be protected from some types of intrusion. More importantly, a storlet in one server cannot cause damage to another server, since in Linux a non-privileged process cannot assume another UID/GID.

3.3 Access Control

Each storlet request is performed on behalf of a specific userid in the storage cloud. If the invocation is on the data path of the storage system (e.g. get or put of an object), the userid is the one issuing the original data request. If the request is not on the data path (e.g. an internal request, possibly event driven), an internal userid can be used.

Access control concerns authentication and authorization to perform certain actions. In the context of the Storlet Engine, there are two aspects of permission:

- *Permission to run the storlet* on a given request on behalf of a given userid. This can be approached in several ways. It may be user/role based. Data objects specified in the request may be associated with a set of rules determining what users and/or storlets are allowed to operate on the object. In fact, when the storlet itself is maintained as an object, the rules of the storlet object can be used.
- *Permission to access an object* in the storage system (get, put, etc.). A storlet should be permitted to access only the objects that the client originating the request is allowed to access. Access rights to objects are enforced by the security mechanisms of the storage system. On access to an object, a storlet needs to present some credentials, usually obtained as a parameter. For example, a token or userid/password in Swift.

Permission to run a storlet must be enforced before the storlet is invoked. Once the storlet starts handling a request, it is assumed that the storlet is authorized for the request. A typical storlet is performed on the data path of the client's original request to the storage system. The storlet is invoked after the storage system has already authenticated and authorized the client's request.

4 Performance Study

4.1 Test Goals

The performance study attempts to answer the following questions:

1. *What performance benefits can be derived by wrapping a function as a storlet?* To answer this question we compare the performance of alternative storlet wrappings against their performance when equivalent functions are run outside of the storage system. The comparison are taken while a fixed workload is running concurrently in the background to simulate the environment in which storlets will run.
2. *How storlets affect system performance?* To answer this question we change our view point and examine the workload's performance when running concurrently with storlets or equivalent functions running outside of the storage system.
3. *What is the performance implications of running storlets on the storage system's interface nodes as oppose to running them on the local storage nodes?*
4. *What host resources are most affected by storlet?* To answer this question we examine host memory utilization, load, cpu utilization and swap utilization.
5. *Do the performance issues (described above) change when the storage system is a private cloud accessed over a high speed internal network or a public cloud accessed over the WAN?* To answer this question we created a Swift test bed with a 10Gb network to simulate a private cloud and add delays to the incoming packets on the client host running the storlets or equivalent function to simulate WAN access.

4.2 Fixity Test Set Results Overview

This subsection compares the performance of alternative fixity storlet wrappings with its counterpart equivalent application. We denote by *treatments* the various alternative storlet wrappings or equivalent functions. The fixity test set includes treatments that calculate a data object's digest, using MD5 and SHA256, and returns the results to the client. We chose fixity as the subject of the performance test since it allowed us to easily compare an eqivalent function running inside and outside the storage system while varying the object size and number of concurrent threads. The different treatments are described below:

– *fixityStorletAtInterfaceNode* : FixityStorlet run on interface node
– *fixityStorletAtLocalNode* : FixityStorlet run on local node
– *fixityAppWithStorletInfrastructure* : The fixity application runs on the treatment engine where it calculates the data object's digest. Even though the fixity application is not a storlet it is run while the storlet infrastructure is installed in the storage system.
– *fixityAppWithOutStorletInfrastructure* : The fixity application runs on treatment engine, but the storage system is in its native form without the storlet infrastructure. This treatment is included so we can evaluate the additional overhead associated with the storlet infrastructure and compare it with the native storage system performance.

Figure 2 illustrates the relationships between the different fixity treatments' response times. We observe that the fixityStorletAtLocalNode's response time is consistently better than the two fixityApp treatments' response time. While the fixityStorletAtInterfaceNode response time is the worst. When we compare

fixityAppWithStorletInfrastructure against the fixityAppWithoutStorletInfrastructure treatment we also observe that there is overhead associated with the storlet infrastructure but that it is not significant.

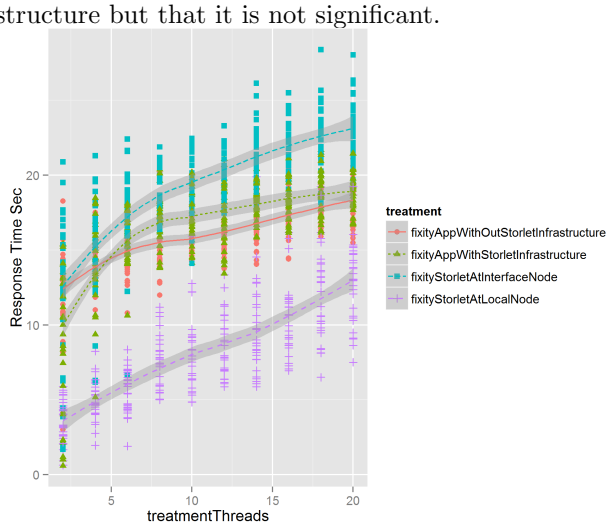

Fig. 2: Fixity treatments for MB size objects response time with less than one millisecond latency between treatment host and interface node.

4.3 Fixity Test Conclusions

1. *What performance benefits can be derived by wrapping a function as a storlet?* The test demonstrates that storlets that run on the local node and process megabyte size objects perform better than the equivalent functions running outside of the storage system.
2. *How storlets affect system performance?* The test demonstrates workloads that overlap storlets that run on the local node and process megabyte size objects perform better than those that overlap equivalent functions running outside of the storage system.
3. *What is the performance implications of running storlets on the storage system's interface nodes as oppose to running them on the their local storage nodes?* storlets that run on the interface node and process megabyte size data perform worse than storlets that run on the local node. Using the interface node to run storlets is only recommended when the storlet does not require much data handle as is the case of the Distributed Storlet.
4. *What host resources are most affected by storlet?* There is a correlation between memory utilization and load at the interface node and the resulting performance.
5. *Do the performance issues (described above) change when the storage system is a private cloud accessed over a high speed internal network or a public cloud accessed over the WAN?* The performance improvements attributed

to storlets that run on the local nodes is amplified as the latency increases between the client host and storage system's interface node.

More details of the performance study can be found in the expanded version of this work [4].

5 Discussion and Conclusions

5.1 Related Work

Discovering new insights from the vast amount of the existing biomedical data is an on-going goal [5]. Challenges towards that goal include efficient processing of large biomedical data sets [6] and privacy issues [7]. The Storlet Engine can support these two challenges by processing data within the storage.

Stored procedures and active databases are at a high level analogous to storlets in object storage. However, object storage is meant for large blobs and provides eventual data consistency while databases are meant for tabular data and provides strong data consistency. Consequently, the Storlet Engine need to confront with different challenges and solve them with distinct approaches.

Efficient execution of data-intensive applications was investgated for many years e.g. [8]. As the data objects become larger and distant, there is an increasing focus on performing computations close to the data. Ceph [9] is a free software storage platform designed to present object, block, and file storage from a single distributed computer cluster. Ceph has the ability to add class methods which enables computations in the storage. However, these class methods cannot run in a sandbox for now. In contrast, our Storlet Engine runs code within a sandbox, which provides better security.

OpenStack Savanna [10] attempts to enable users to easily provision and manage Hadoop clusters on OpenStack. A similar technology is Amazon Elastic MapReduce (EMR) service that provides elastic Hadoop cluster on Amazon cloud. In both technologies, the data is still transferred from the object storage to the compute nodes, unlike our Storlet Engine that performs the compute in the object nodes of the cloud object storage.

ZeroVM [11] provides a virtual server system that can run close to your data. It is an open source lightweight virtualization platform based on the Chromium Native Client (NaCl) project. It claims to be fast, secure and disposable but the use of NaCl requires customized versions of the GNU tool-chain, specifically gcc and binutils. Consequently, existing code need to be recompiled for the NaCl tool-chain which is sometimes non-trivial. In contrast, in our Storlet Engine recompilation is not required, and existing code can be executed unaltered.

5.2 Conclusions and Future Work

The paper presented Storlet Engine, an environment supporting computations within storage system. The idea is to extend the traditional role of object storage as a repository for data, by exploiting the computing resources of storage nodes to run computation modules – storlets – on the data close to where it resides.

We described the architecture and key features of the system, and discussed the implementation of storlets for biomedical processes such as ROI Extraction, Cell Detection, Cohort Identification. Some storlets for these business processes were implemented in the context of the ENSURE and ForgetIT EU projects. We also conducted a performance study, evaluating the Storlet Engine within OpenStack Swift prototype.

There are multiple opportunities for further work and research. Our future plans include exploring the usage of storlets for genomic workflows and for medical imaging analytics of various modalities such as ultrasound, mammography, CT. Some open questions in this area include: what computations in biomedical processes should be executed as storlets? What parameters influence this decision? Can the decision be taken transparently to the business process developer and user? Exploring the merit of a storlets marketplace is a another direction, and in particular identifying biomedical use cases and applications. Finally, for this evolving methodology, a standardization effort is a long term goal.

Acknowledgments. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement 270000 and under grant agreement 600826.

References

1. Factor, M., Naor, D., Rabinovici-Cohen, S., Ramati, L., Reshef, P., Satran, J., Giaretta, D.: Preservation DataStores: Architecture for preservation aware storage. In: MSST 2007: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies, San Diego, CA (September 2007) 3–15
2. Rabinovici-Cohen, S., Marberg, J., Nagin, K., Pease, D.: PDS Cloud: Long term digital preservation in the cloud. In: IC2E 2013: Proceedings of the IEEE International Conference on Cloud Engineering, San Francisco, CA (March 2013)
3. Rajaraman, A., Ullman, J.: Mining of Massive Datasets. Lecture Notes for Stanford CS345A Web Mining. (2011)
4. Rabinovici-Cohen, S., Henis, E., Marberg, J., Nagin, K.: Storlet Engine: Performing Computations in Cloud Storage. Technical Report, IBM Research – Haifa (2014, to be published)
5. Shahar, Y.: The elicitation, representation, application, and automated discovery of time-oriented declarative clinical knowledge. In: ProHealth/KR4HC. (2012)
6. Cooper, L., Carter, A., Farris, A., Wang, F., Kong, J., Gutman, D., Widener, P., Pan, T., Cholleti, S., Sharma, A., Kurç, T., Brat, D., Saltz, J.: Digital pathology: Data-intensive frontier in medical imaging. Proceedings of the IEEE **100**(4) (2012)
7. Le, X., Wang, D.: Neuroimage data sets: Rethinking privacy policies. In: HealthSec. (2012)
8. Rabinovici-Cohen, S., Wolfson, O.: Why a single parallelization strategy is not enough in knowledge bases. J. Comput. Syst. Sci. **47**(1) (1993) 2–44
9. Weil, S., Brandt, S., Miller, E., Long, D., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: OSDI 2006: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. (2006)
10. OpenStack Savanna. URL `https://wiki.openstack.org/wiki/Savanna`
11. ZeroVM. URL `http://zerovm.org`