Extending Cloud-based Object Storage with Content Centric Services

Michael C. Jaeger[1], Alberto Messina[2], Spyridon V. Gogouvitis[3], Elliot K. Kolodner[4], Dimosthenis Kyriazis[3], Enver Bahar[1] and Uwe Hohenstein[1]

[1]*Siemens AG, Corporate Technology, D-80200 Munich, Germany*

[2]*RAI Radiotel. Italiana, Centre for Research and Technological Innovation, Corso Giambone 68, I-10135 Turin, Italy*

[3]*National Technical University of Athens, Heroon Polytechniou, 15773 Athens, Greece*

[4]*IBM Haifa Research Lab, Haifa, 31905, Israel*

Keywords: Cloud Computing, Cloud Storage, Metadata Management, Object Storage.

Abstract: Content centric storage refers to a paradigm where data objects are accessed by applications through information about their content, rather than their path in a hierarchical structure. Applications are relieved from having knowledge about the data store organization or the place in a (physical) storage hierarchy. Instead, applications can use metadata associated with objects in order to to query for the desired content. In this paper, we explain the new functionality added to our first version of the content centric storage (Jaeger et al., 2012): We present a new REST API for the management of relations and the ability to use schemas for enforcing metadata. The need for such a content centric storage is presented with examples from the media production domain.

1 INTRODUCTION

There are an increasing number and a variety of applications that face a growing need for digital storage. For example, there are many media authoring applications that work with video files beyond high definition video: Ultra HD represents the next generation video format which defines sizes up to 7680 by 4320 pixels. Archives for large virtual machine disks also require growing quantities of storage due to the virtualization trends in today's enterprise IT. Moreover, mobile devices are producing multimedia content in an exponentially growing manner.

All this contributes to an ever increasing number and size of storage objects, which mostly are large and unstructured. New challenges arise not only from the numbers and sizes of objects, but also through distributing data in order to increase fault tolerance and the availability of data anytime and anywhere, by any device. To satisfy applications, storage systems must be capable of handling large objects or files and/or a large number of files. This includes the ability to easily ingest content of any type, to quickly find the desired content, and to smoothly access the content through any desired device. This functionality extends the plain data storage features offered by cloud storage providers or database systems today.

A common way to handle large content is to put them into files and to organize them in a hierarchical structure. This enables a user to navigate the hierarchy. However, when the amount of data gets huge, it becomes more and more difficult to set up an appropriate hierarchy that provides flexible search options with a acceptable performance for access.

We present in this paper a new approach to content centric storage in which the user is not restricted to organizing his content in hierarchies. Rather, the user describes the content through metadata and then accesses the content based on its associated metadata. Moreover, the storage system itself is also able to derive metadata from usage statistics or access mechanisms. The basis for this approach to content centric storage are efficient mechanisms to automatically create or ingest and later retrieve any kind of metadata about the content, which is the entry point to objects.

Our goal is to provide similar functionality in a more generalized form, in particular that extends the scope to any type of data, not just videos. We expect that all kinds of data become ever more valuable since more and more areas of life are touched by all types of data, no matter whether text, pictures and moving images with updates from sensors, mobile clients etc. Nowadays, handling and using metadata in a conventional data object storage system is often restricted (for example, not offering support for queries).

The EU project VISION Cloud (Kolodner et al., 2011) aims at developing a cloud storage system that allows for efficient and federated storage of all types of content-centric data. In contrast to public cloud offerings such as Amazon S3 (Amazon Web Services, 2012a), Microsoft Blob Service (Microsoft Corporation, 2012) or specific hardware appliances, VISION Cloud stresses supporting metadata flexibly and as an integral part of the storage. VISION Cloud supports private cloud installations and does not require the use of specialized hardware.

Overall we present the motivation, the requirements, and the design for a metadata-enhanced large storage system, called content centric storage in detail. The structure is as follows: In Section 2, we introduce a motivating scenario and derives the corresponding requirements. In Section 3, we provide an overview of our approach to content centric storage. In Sections 4 and 5, we describe our solutions for two aspects of content centric storage, relations and metadata schema, respectively. In Section 6we present a request throughput evaluation and its results. Finally, we explain related work in Section 7 and we conclude the paper in Section 8.

2 THE MEDIA PRODUCTION AND BROADCASTING USE CASE

One of the business domains in which metadata-based access is crucial, is that of media production and broadcasting. In this environment, the ability to search and retrieve objects from an archive for new productions constitutes a key enabler. Media Asset Management Systems are the systems in charge of this task. However, very often the focus of these systems is to provide separate indexing and retrieval functionality from the actual storage. This results in limited or no possibility to integrate systems that operate on the same content, and in the necessity to introduce complex metadata import/export/adaptation modules, which in most cases are not lossless from the information point of view.

An approach that treats metadata as an integral part of the content, instead, has the benefit of making media content items more re-usable across systems and applications, since all its informative content (metadata) is natively attached to them. Metadata can be of several types: technical (e.g., related to the parameters that describe the encoding of the media), descriptive (e.g., about the content of pictures, the people that took part), administrative (e.g., costs related to production), content-related (e.g., color related features or loudness) and relational or structural (e.g., information about interlinked media items).

Specifically with respect to the last metadata type, it should be pointed out that the ability to retain the relations between content items allows for more powerful access to many aspects concerned with content, e.g., derivation history, editorial versions, equivalent content items, related items, and enrichments. The data items, scenes, videos, audio tracks, different sequences etc. have important relations between them that should be supported by the storage. A producer searching an archive requires information about which audio tracks belong to which video tracks, or, which material belongs together because it represents a sequence of shots, etc. Therefore the storage needs to support relations between the items. Finally, metadata often already exists in different forms or as part of different domain specific file formats. This metadata is very important for searching across the content. Therefore, the metadata capabilities of the storage must support the metadata data models already established in the media domain.

3 GENERAL OVERVIEW OF THE CONTENT CENTRIC APPROACH

The first version of our approach was presented in (Jaeger et al., 2012). In this paper, we present in the subsequent three sections new functionality and findings based on this work. The cornerstones of our approach did not change: Our work focuses on the content centric functionality not on the implementation of a storage system. Therefore, the implementation is separate from the data object store. It can be combined with several "classic" data object stores. For our development, we work with either the storage of the project VISION Cloud or the CouchDB document database (Anderson et al., 2010).

The anticipated deployment for the content centric service is a cloud-deployment. When the service components are combined with a data object store or CouchDB, the components are deployed on every node where the actual storage can be requested. It acts as a wrapper to the underlying storage. The approach is to store metadata in the form of key-value pairs with data objects and implement the additional functionality on top of it. A REST-based service implementation is provided, similar to existing storage services, such as Amazon S3 or the Microsoft Azure

Blob Service. In fact, the content centric REST interface is oriented towards the Cloud Data Management Interface Version 1 from the SNIA consortium (Storage Networking Industry Association (SNIA), 2011).

The general technical structure features a Java servlet application with a three tier design: An interface layer to decode the REST requests, a logic layer to perform the actual request, and a storage client layer that allows for a pluggable access to different underlying storage system. The handling of REST requests is based upon the Jersey framework (Java.net, 2012). Jersey is a javax-rs implementation and part of the Glassfish project. Every application can store metadata with data objects in the underlying storage service. The content centric service provides advanced capabilities on top. Consider the requirements from the media use case to support relations: For example, different media assets can form a sequence. This sequence information should be also placed at the media assets in the storage. However, this implies a systematic approach for expressing relations. Otherwise, different applications develop their own proprietary way of expressing the relation information and then reuse of this metadata becomes difficult. Thus, in our approach is domain-independent, and applications, e.g., media applications, can concentrate on managing the relations with a stable API.

Figure 1 shows the subcomponents of the content centric service. Applications can access this component via the top layer, the access layer. The separation from the actual implementation, placed one layer below, allows for supporting different interface technologies in future, if necessary. All implementation code accesses an abstract storage adaptor, which is placed in the third layer, the storage layer. Currently, the CouchDB and the VISION Cloud storage are primarily supported. Additional storage services can be added, if an adaptor is implemented that adheres to the abstract storage interface. The Figure 1 also exposes the functional modules: An upload service allows for the batch import of existing metadata for multiple objects. As a configuration, a mapping file is uploaded that explains how the information in the XML document is translated into the key-value metadata pairs. This service allows to import metadata for data objects from existing sources. A schema service allows for uploading a metadata schema which is used for checking the metadata of data objects. The CDMI Storage Service follows the Cloud Data Management Interface (CDMI) for normal metadata operations as well as adding additional query capability. In addition, this part implements the REST interface for the relations. CDMI is a standard from the Storage Networking Industry Association (Storage Networking

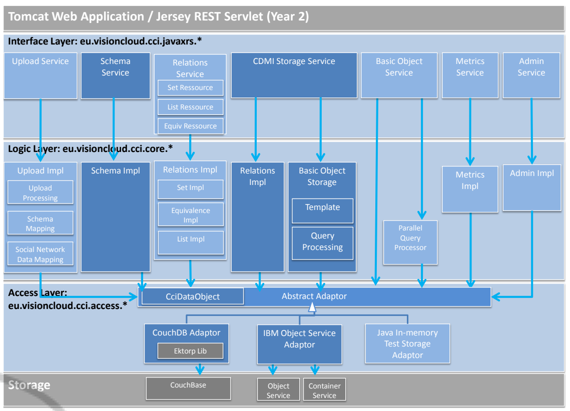

Figure 1: Overview of the Content Centric Service Component.

Industry Association (SNIA), 2011) The Basic Object Service allows for basic object and metadata creation, update and delete methods. It represents a simplified version of CDMI Storage Service. The Metrics Service offers an interface for addint metrics to data objects as metadata.

4 RELATIONSHIP CONCEPT

The support for relations between entities is fundamental for the querying capabilities provided by content centric service. The first content centric service implementation (Jaeger et al., 2012) supported the following relations: *a*) a set relation, *b*) a list relation and *c*) an equivalence relation. In a nutshell, a set relation can mark several content items that belong together. A list relation adds ordering information to the set property. If two different data objects actually contain the same content, maybe at different video screen resolutions for example, an equivalence relation can be used. However, the first version required a REST service implementation for each relation, which implies that for each new relation, a new service must be implemented and deployed. Here, we present the new more general approach which allows for new relations to be implemented in the future. The relation information is entirely stored as metadata of the data objects. More relations are required, for example if we consider the idea of storing relations is similar to the Subject-Predicate-Object approach in the Resource Description Framework (RDF, (World Wide Web Consortium, 2004)).

4.1 Storing Relation Metadata in a Key-value Storage

An important design decision of this implementation is to avoid separate data entities in the store that contain information about relations. Our goal was to avoid distributing metadata over several entities, because the physical storage location of two different data entities can be distributed in a cloud-oriented architecture. Then, access latencies would be likely. Instead, we place information about the relations directly in the data object's metadata. The implementation makes use of metadata keys and values. The use of key and values in a metadata of a data object bears some issues: For example, a set membership can be defined using key-value metadata as `key:"set", value:"somesetid1"`. This raises the problem that in order to express two set membership for the same data object at once, the value needs to contain the set ids of both sets. For example: `key:"set", value:"somesetid1,somesetid2"`. However, our underlying key-value storage may not support wildcard queries (i.e. querying for items that belong to set id 2: "*somesetid2*"). Therefore the next approach is to denote a relation membership with the key of a key-value pair only, for example: `key:"@setid-somesetid1"`. The set is represented by a key $@setid - somesetid1$, i.e., each member of the set contains the same key $@setid - somesetid1$. Here, "@" denotes that this metadata key is not explicitly set by the user. And the "setid"-part defines that this key refers to a set relation. Searching for all members of a set means that a query can just search for data objects where the key $@setid - somesetid1$ exists. To denote a second membership for the same object, another key can be easily added to the metadata catalogue of a data object.

This approach works if the storage system allows for searching for keys that are in use by a prefix. Then, the implementation can query for keys beginning with $@setid - *$. The sets actually in use (or already assigned set ids) can efficiently be queried, so that an application does not need to keep track of the sets in use. An issue arises, if the underlying storage system does not support removing metadata items from a data object's metadata catalogue. In a distributed versioned storage, consistency problems can be solved if metadata keys are not erased but invalidated at some version while the values can be changed. However, if the key remains in the metadata catalogue of a data object once it was created, the membership of a relation cannot be removed. As a solution, the value of the referring key can be toggled between true or false denoting whether the relation membership is active or not. For example, a set membership declaration is represented through key-value metadata as: `key:"@setid-somesetid1", value:"true"` in order to denote that set membership for set *somesetid*1 is currently valid for this data object.

While these issues refer to the approach for storing relations, the design of the REST API has also some issues. The general way of integrating the relations into the hierarchy of containers and objects was to see the relations as a concept valid within one container. And, the relations refer to one or more data objects. Accordingly a natural way of expressing a resource path in the REST sense is for example `PUT /container/someset1id/myDataObject1` to define the set relation membership for an existing data object. Or, the request path `GET /container/someset1id/` requests the objects of the relation. We introduced a new Content-Type header value for the HTTP requests in order to distinguish a data object resource path from a relation resource path. Therefore, requests on relations use the Content-Type $cdmi - relation$. Note that this is aligned with the other content types of the CDMI standard from SNIA, such as $cdmi - query$ or $cdmi - container$.

Besides the resource path, the content for the Content-Type $cdmi - relation$ is used to place or request information about the relation. Existing industry standards already cover general schemes for expressing relations. Popular examples are the MetaObject Facility (MOF, (Object Management Group (OMG), 2006)) by the OMG, which covers relations between entities in software or data models or the RDF. As related work, also Zygmuntowicz has discussed how to express relations in a key-value store (Zygmuntowicz, 2010). So continuing with the previous example expressing set membership for the set with the id *somerelid*1 at a data object looks like:

```
@reltype : set
@relid-somerelid1 : true
```

However, this way of expressing it bears the conflict that only one membership per relation is allowed, otherwise it would be unclear what the relation type refers to. Therefore we use the following metadata scheme instead:

```
@reltype-somerelid1 : set
@relid-somerelid1 : true
```

Continuing, a list needs a number, or a rank. And this brings us to RDF where we have a triple consisting of a subject, a predicate and an object:

```
@reltype-somerelid1 : list
@relid-somerelid1 : true
@relobj-somerelid1 : 3
```

The above example denotes that the data object (to which the metadata is attached) is a member of a list with Id *somerelid*1 at index/rank/position 3. This directly corresponds to an RDF scheme for data properties of objects: (1) A subject which is data object that has the metadata information attached, (2) a predicate, which is a combination of @reltype value and encoded relation id in attribute, and (3), the object, which is the value of the $@relobj - *attribute$. Staying with the RDF comparison, a different situation holds when we want to represent object properties, i.e., relations that hold among objects. In this case the following scheme should be applied:

```
@reltype-somerelid1 : list
@relid-somerelid1 : true
@relobj-somerelid1 : 3
@relcontext-somerelid1: someobjid
```

Above, the ”relcontext” specifies the object in the context of which this relation is valid. We consider a concrete example. If we store both a PhotoCollection object and the list of its contained photos, we want to represent: *a*) the position of all the photos in the list; *b*) the fact that the list is valid in the context of a specific photo collection. The following example states that the photo ”myphoto101” is in position 3 of the list named ”contained_photos” for the parent object ”PhotoCollection1”.

```
id : "myphoto101"
@reltype-contained_photos : list
@relid-contained_photos : true
@relobj-contained_photos : 3
@relcontext-contained_photos: PhotoCollection1
```

This solution allows to search for specific relations in which an object is involved without filtering out the part of the key which encodes the context object. One could search for the presence of ”@relid-contained_photos” as a key to know what photos are in any possible collection. By making this extension, we als extend the RDF relation model, by introducing a ”role specifier” represented by the attribute ”@relobj-somerelid”. Please note that we provide an example application REST request flow using the relations in the appendix.

4.2 Application in the Media Use Case

As anticipated, media production processes are metadata-eager. This is true both from the descriptive and the structural perspective, as outlined in Section 2. In this Section, we provide a couple of examples of how the functionalities of the content centric relations have been employed in this domain. The MXF (Material eXchange Format, (SMPTE, 2011)) is the master reference nowadays for professional media production. MXF files are metadata-rich both in descriptive and in structural terms, being able to represent a full range of possible operational patterns with media files and to carry user-defined and standard sets of metadata. In one of the VISION Cloud experiments an MXF file is uploaded starting with an XML metadata import descriptor using the upload service of the content centric service. As part of the subsequent metadata imports, also two lists of objects are created: one that groups the video frames together and another that groups the audio clips from the MXF file import. These lists allow one later to respectively access the individual video frames and audio clips of the file with plain HTTP GET requests:

```
GET /container/Materialid_videoframes
GET /container/Materialid_audioclips
```

In this example, ”Materialid_videoframes” is the list id that holds objects of the container named ”container”.

A significant functional progress w.r.t. the previous example is represented by the integration with external enterprise-level media production applications like cross-media aggregators for news (Messina et al., 2011). This import operation involves complex multimedia data rather than just plain MXF files. An aggregation platform (Messina et al., 2011) performs complex analysis operations on data sources which result in hybrid aggregations of Web and television resources around automatically detected topics. These aggregations form the basis for further editorial work performed by journalists who want to enrich or follow a detected story. It is therefore very important being able to losslessly transfer the information structure that links together the resources of a determined topic in the production environment. Using a set relation, the items are grouped together. The set relation is created at import of the metadata using the Upload service. From then, the items belonging to the set can be accessed by HTTP GET requests:

```
GET /container/AggregationId_webnews
GET /container/AggregationId_tvnews
GET /container/AggregationId/TVClipId_keyframes
```

The first call returns the web articles included in a specified aggregation (topic), the second call returns the television news items of the same aggregation, the third call returns the list of key frames of a specific television resource of the same aggregation.

5 METADATA SCHEMA CHECKING

The metadata items associated with objects in the underlying storage service, e.g., CouchDB and the VISION Cloud object service, are schema-less. That means the number of keys and their designation is flexible. Arbitrary metadata keys can be attached to each data object. This can be useful, as it provides flexibility. However, for certain use cases, some control on the metadata use is desired. For example, an application would like to ensure that uploads of data objects comply with a certain schema in order to enable further processing of this data. An application must be able to define a schema for the metadata keys to be used. When such a schema exists, creating or changing metadata should be checked for compliance. As a first step, such a check should ensure that certain metadata keys are actually present for a data object. A challenge arises when considering a classic XML Schema definition for metadata information derived from standard data formats. An example schema that is used to define the metadata items to check for looks like the following listing:

```
...<xs:element name="shiporder"> <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType> <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" .../>
            <xs:element name="city" .../>
            <xs:element name="country" .../>
         </xs:sequence> </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded"
              minOccurs="0">
        <xs:complexType> <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note".../>
            <xs:element name="quantity" type="xs:pos...Int"/>
            ... </xs:sequence> </xs:complexType>
      </xs:element> </xs:sequence>
    <xs:attribute name="orderid" type="xs:string"/>
  </xs:complexType> </xs:element></xs:schema>
```

In this example, the data is structured hierarchically: an ”orderperson” consists of several subelements. In this case, ”item” and ”orderperson” are elements that contain several subelements and also form together the complex element ”shiporder”. In contrast, the underlying key-value storage offers only metadata key-value pairs. Therefore, the implementation of the schema check must also provide a means for expressing the hierarchy of XML tags in the hierarchy-agnostic metadata format. Consider the following hierarchical JSON example:

```
{ metadata={ "shiporder"={  "orderperson"={
        "name"="J.A.",
        "address"="Flat 3B, 3 Hans Crescent",
        "city"="London SW1X 0LS",
        "country"="UK" },
      "item" ={
        "title"="...", "note"="...",
        "quantity"="...", "price"="..."  },
      "orderid"="12345678"
    } } }
```

If the hierarchy remains without cross references, which is the assumption made, then the equivalent representation using flat key-value metadata looks as follows:

```
{  "shiporder.orderperson.name"="J.A.",
    "shiporder.orderperson.address"=
        "Flat 3B, 3 Hans Crescent",
    "shiporder.orderperson.city"=
        "London SW1X 0LS",
    "shiporder.orderperson.country"="UK",
    "shiporder.item.title"="...",
    "shiporder.item.note"="...",
    "shiporder.item.quantity"="...",
    "shiporder.item.price"="..." }
```

The schema defines the metadata information to be present. Because schemas could be either XML Schema or equivalently use a hierarchical JSON representation, for this schema a representation for a hierarchy agnostic metadata such as that supported in the underlying storage services is created.

5.1 Functionality of the Provided Service

We implemented a REST service that allows for the upload, listing and deletion of such schemas. It provides the general translation of hierarchical JSON / hierarchical XML into flat keys or key-value pairs for an ”internal schema” and storing them for further processing. The internal representation uses flattened key names where the hierarchy is transformed into a dot-separated chain of hierarchy elements. In our implementation, the ”internal schema” is stored in the metadata section of a data container. Then, also the metadata of the data object must be translated from hierarchical JSON into flat keys. Contrary to the previous point, a schema (not necessarily an XML schema) is uploaded to the VISION Cloud first. Then, the subsequent upload of metadata of a data object's in JSON or XML is translated accordingly and compared with the schema. If the upload of data object metadata does not conform, the request fails (HTTP response code 400). Conformance is defined that the defined metadata keys must exist at metadata creation or metadata modification.

A metadata schema, to which object metadata must conform, is known to VISION Cloud. It is stored at container level of the VISION Cloud storage or at the level of the CouchDB database. Accordingly, every object in a container can be covered by this schema. In addition, the application can also define a schema for a particular mime-type. In this case, the schema check is only applied to objects that have a specific mime-type defined. Note that the application can upload such a schema for a specific mime-type, but this applies to the coverage of a single container as well. The usage flow is as follows:

1. The developer or user defines an application metadata schema, or uses an existing metadata schema using the XML Schema syntax.
2. The developer or user uploads this schema. Either this schema defines the metadata structure for all objects of a container or only for objects with a specific mime-type of a container.
3. The developer or user uploads data objects with metadata or changes metadata. The content centric service checks for conformance with the schema. In the negative case, a response code 400 is returned and the request is not fulfilled.

6 REQUEST THROUGHPUT VALIDATION

The requests to the underlying storage pass through the content centric service component. Therefore, it is important to ensure that this component handles requests in an efficient way and must not represent a bottle neck. In order to verify its efficiency, we have conducted a set of performance tests in order to evaluate the query performance of the content centric service component's layered approach, presented in our previous work (Jaeger et al., 2012). In our recent development, we have performed the validation again, considering more statistical measures in order to also assess the volatility of the results; our validation goals were as follows: (1) If the implementation shows a continuous and repeatable level of query throughput. (2) Check for an acceptable query handling time: How much milliseconds are spent on each individual request? (3) Gain a deeper understanding about how to process queries internally: *a*) work with multiple threads or work in a single thread behavior?, and *b*) when a set of metadata fields is fetched, how does it impact performance if we get all the metadata at once or fetch a couple of individual items?

For this run of tests, we have used the same computer (virtual) hardware and the same test application as in our previous work (Jaeger et al., 2012). This has the advantage that the results are directly comparable. The scenario originates from the media domain and is a test application of the VISION Cloud project, developed by Deutsche Welle (DW). The query is as follows: from a repository of news video material, find all news video assets within a given date range sorted by popularity, grouped by news channels. For the request throughput evaluation of the content centric service component, video material data objects were considered with metadata information attached to them. The size of the metadata, not the data objects, amounts to 450MB in total. These items were stored in a CouchDB server.

Figure 2 is based on Figure 1 and shows the request path in the software stack for this test. The measured time started when the REST request from the test application arrived at the REST service implementation and ended before the request was passed back. This is indicated by the top horizontal line in the Box ”CDMI Storage Service” of Figure 2. Technically, these times correspond to the time from when the Jersey servlet calls the REST service implementation to the point when the service implementation ”returns” to the Jersey servlet. It also includes the query to the underlying storage. For the evaluation,

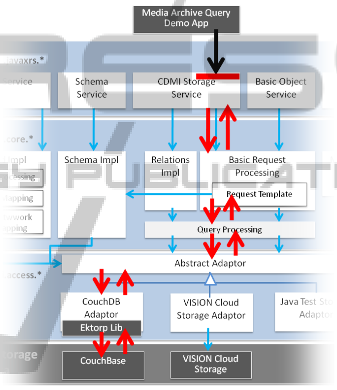

Figure 2: Technical View on the Test Path.

the following cases were investigated:

- Evaluate the request response time on a single core and a multi core machine.
- Evaluate the request response time for three different query date ranges: 1 day, 2 weeks and 4 weeks. Each of the query date range resulted in a different number of data objects that should be part of the result set (30, 663, and 969 data sets respectively).
- For the elements of the result set, additional metadata key-values must be queried. We evaluated the response time when querying for the metadata items in individual queries to the underlying storage (”channel', ”popularity” and ”date”) compared to querying for all of the metadata associated with the data object at once. The average size of metadata for each data object was about 1KB.

6.1 Evaluation Results

We have conducted the tests on two test platforms summarized in Table 1.

Table 1: Overview of the Two Test Setups.

Name CPU	Cores	Speed	L2	RAM	OS	Storage
Virt. 1-core	1/1	2.13Ghz	4MB	8GB	RHEL	500GB
Vir. 4-core	4/4	2.13Ghz	4MB	8GB	RHEL	500GB

For the evaluation, a plain setup of the VISION Cloud software stack was used, disabling services such as authorization and monitoring. Also, the log level was set to *fatal* in order to avoid efforts for console I/O. It should also be noted that schema checks did not apply, because no metadata was modified or created. The measurements were analyzed with basic statistics, such as the deviation and a 95% confidence interval. The results are presented in Table 2 for a single core deployment of the test setup and in Table 3 for a quad core deployment of the test setup.

In both tables, the first column specifies the test configuration: $a)$ using either a single threaded application or using a thread pool, and $b)$ the setting of querying each metadata value in individual REST requests or querying the entire metadata catalogue of the data object in one REST request. The next column lists the resulting number of requests to the underlying key-value storage which the content centric service has used in the test. The measured times ("mean") are given in milliseconds and represent the average mean of 20 runs. For the mean calculation the longest and shortest run of the measured execution times have been omitted. The last column considers the overall run time of the application requests and the number of resulting requests issued to the underlying storage. From these two values the fractional response time per each request to the underlying key-value storage is given.

Table 2: Second Test Run Results: Single Core Virtual Machine (times in milliseconds).

	Individual	Req. Count	Av.Tot. (msec)	Std Devia.	95% Conf.	Single msec
Single Thread	Separate Metadata Calls	90 1989 2907	823,8 12761,8 18310,3	117,8 1189,1 273,0	42,1 425,5 97,7	9,2 6,4 6,3
	Combined Metadata Calls	30 663 969	320,6 6632,6 9409,9	26,7 59,8 126,9	9,5 21,4 45,4	10,7 10,0 9,7
Thread Pool	Separate Metadata Calls	90 1989 2907	431,8 8807,6 12842,6	24,6 207,4 200,7	8,8 74,2 71,8	4,8 4,4 4,4
	Combined Metadata Calls	30 663 969	226,2 4596,4 6631,2	12,0 115,1 125,2	4,3 41,2 44,8	7,5 6,9 6,8

These results show the following w.r.t. to our evaluation goals:

- Continuous and repeatable level of query processing throughput: The confidence intervals and the deviations let us conclude that response times show low deviation. It must be noted that the different measurements on one machine were conducted without restarting the components.
- Check for an acceptable query handling time: When performing the largest setup, the average response results in a few milliseconds. Since the underlying storage is also queried by using a REST call, which is also included, this represents an acceptable value.
- Gain a deeper understanding about how to process queries internally: (a) Working with threads or work in a single thread behavior: The evaluation shows that parallel queries to the same storage results in lower response times. (b) Get all the metadata at once or fetch individual items: Given three individual metadata items to query, lower response times were yielded when querying the entire metadata set from the storage and filter the three relevant values locally.

Table 3: Second Test Run Results: Quad Core Virtual Machine (times in milliseconds).

	Individual	Req. Count	Av.Tot. (msec)	Std Devia.	95% Conf.	Single msec
Single Thread	Separate Metadata Calls	90 1989 2907	836,8 13735,0 20276,4	54,4 785,5 963,8	19,4 281,1 344,9	9,3 6,9 7,0
	Combined Metadata Calls	30 663 969	358,0 7790,4 11344,2	32,5 234,0 420,7	11,6 83,7 150,6	11,9 11,8 11,7
Thread Pool	Separate Metadata Calls	90 1989 2907	172,2 3217,1 4574,4	13,3 158,1 99,2	4,7 56,6 35,5	1,9 1,6 1,6
	Combined Metadata Calls	30 663 969	101,1 1823,0 2546,0	12,5 102,2 117,4	4,5 36,6 42,0	3,4 2,7 2,6

7 RELATED WORK

One of the known commercial solution is Amazon S3 (Amazon Web Services, 2012a). S3 allows the storage of large objects along with a set of metadata values. Nevertheless, there is no support for finding specific objects based on their metadata. Moreover, there is no support for user metadata to follow a specific schema. Some of this functionality can be achieved by storing object metadata externally through Amazon's SimpleDB (Amazon Web Services, 2012b). Such a solution relies on building proper functionality on top of S3 and simpleDB to ensure consistency and is not provided inherently. The Microsoft Windows Azure platform provides storage services in the form of the Blob Storage service (Microsoft Corporation, 2012), the Table service, and the Windows Azure Drives service which provides single volumes (NTFS VHD). Similar to S3, objects can also have metadata, but the platform does not allow for the advanced queries. Other storage cloud solutions such as (Google, 2012) or (Rackspace, 2012) have the similar characteristics. EMC Atmos (EMC Corporation, 2012a) is a storage platform that can be used to build private or public clouds. EMC Atmos allows queries by metadata key but it does not provide the schema checking support or setting relations between objects for example.

Object database such as Versant Store (Versant, 2012), Objectivity/DB (Objectivity, 2012) or VelocityDB (Velocitydb.com, 2012) could be also used. While fault-tolerance, high-performance and scalability can be achieved, this is accompanied by a high technical effort. Moreover, the maximum size of the stored objects is limited compared to cloud offerings. Apache HBase (Apache Foundation, 2012b) is a distributed data store built on top of Hadoop/HDFS (Apache Foundation, 2012a). The solution is inherently scalable; but it is more oriented towards table storage and lacks features needed to enable content centric access. VISION Cloud, through the content centric service, allows accessing stored objects, the content they hold, their relationships and the interpretation of their metadata.

Content-addressable storage (CAS) systems, such as EMC Centera (EMC Corporation, 2012b) and Venti (Quinlan and Dorward, 2002), assign a unique name to objects that is produced based on the object contents. This makes the location of the object irrelevant, since it can be retrieved solely based on its unique name. CAS systems are tailored for archiving data and are therefore not suited for general purpose. A notable research effort is CIMPLE (Delaet and Joosen, 2009). In CIMPLE, every content item is represented with a unique key that is calculated from the content and it is associated with a set of attributes that contains information regarding the owner, the key used to create the unique ID etc. Moreover, metadata can be associated with every item that can be used to carry out search operations. Metadata is expressed through RDF, queried through SPARQL and stored in a separate database. While some of the ideas of CIMPLE are also applicable in our effort, there are differences. First of all, it is a stand-alone system whereas our solution is part of a complete cloud storage solution. Secondly, CIMPLE relies on a separate database to handle metadata, which is in direct contrast to our data model, where data and metadata are treaded as one entity. Moreover, the solution also not covered scalability issues, as acknowledged by the authors.

8 CONCLUSIONS

The approach of content centric storage enables an application or user to access storage items based on their content, rather than a path in a hierarchical directory tree. In particular, an application or user describes the content of a storage object through metadata associated with the object and then finds or accesses a storage object based on its associated metadata. In this paper, we extend our work on content centric storage, including a generalized approach to expressing relations between data objects, metadata schema checking and a performance evaluation.

Our earlier approach to implementing relations was not easily extensible; each new relation required a new implementation. In this paper, we describe a unified and extensible scheme for the metadata fields used to express relations. We also demonstrate how to apply it to express the relations required by the media production process. Given the number and size of metadata fields that can be associated with an object, we introduce metadata schema to describe and enforce the use of metadata. The metadata schema checker allows the application to input hierarchical metadata; it flattens the hierarchical metadata to a representation expected by the lower level storage service. Finally, we did a performance evaluation to check implementation alternatives and the request throughput in order to make sure that the implementation meets the expectations from a cloud computing enviroment: scalability and performance.

ACKNOWLEDGEMENTS

The research leading to these results is partially supported by the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement nr. 257019 - VISION Cloud Project.

REFERENCES

Amazon Web Services (2012a). Amazon Simple Storage Service. http://aws.amazon.com/s3/.

Amazon Web Services (2012b). Amazon Simpledb. http://aws.amazon.com/simpledb/.

Anderson, J. C., Lehnardt, J., and Slater, N. (2010). *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition.

Apache Foundation (2012a). Apache Hadoop. http://hadoop.apache.org/.

Apache Foundation (2012b). Apache HBase. http://hbase.apache.org/.

Delaet, T. and Joosen, W. (2009). Managing your content with CIMPLE - a content-centric storage interface. In *IEEE 34th Conf. on Local Computer Networks, 2009. LCN 2009*, pages 491 –498.

EMC Corporation (2012a). EMC Atmos. http://www.emc.com/storage/atmos/atmos.htm.

EMC Corporation (2012b). EMC Centera. http://www.emc.com/products/family/emc-centera-family.htm.

Google (2012). Google Cloud Storage. http://cloud.google.com/products/cloud-storage.

Jaeger, M. C., Messina, A., Lorenz, M., Gogouvitis, S. V., Kyriazis, D., Kolodner, E. K., Su, X., and Bahar, E. (2012). Cloud-based content centric storage for large systems. In *Fed. Conf. on Computer Sc. and Information Systems - FedCSIS 2012, Wroclaw, Poland, September 2012*, pages 987–994.

Java.net (2012). Java.net / glassfish: Jersey project page, accessed in september 2012 at http://jersey.java.net/.

Kolodner, E. K., Tal, S., Kyriazis, D., Naor, D., Allalouf, M., Bonelli, L., Brand, P., Eckert, A., Elmroth, E., Gogouvitis, S. V., Harnik, D., Hernández, F., Jaeger, M. C., Lakew, E. B., Lopez, J. M., Lorenz, M., Messina, A., Shulman-Peleg, A., Talyansky, R., Voulodimos, A., and Wolfsthal, Y. (2011). A cloud environment for data-intensive storage services. In *CloudCom*, pages 357–366.

Messina, A., Montagnuolo, M., Di Massa, R., and Borgotallo, R. (2011). Hyper media news: a fully automated platform for large scale analysis, production and distribution of multimodal news content. *Multimedia Tools and Applications*.

Microsoft Corporation (2012). Microsoft Azure Blob Service API. http://msdn.microsoft.com/en-us/library/dd135733.aspx.

Object Management Group (OMG) (2006). Meta Object Facility (MOF) Core Specification Version 2.0, 2006, http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

Objectivity (2012). Objectivity DB. http://www.objectivity.com/pages/objectivity/default.asp.

Quinlan, S. and Dorward, S. (2002). Venti: A New Approach to Archival Storage. In *FAST'02*, pages 89–101.

Rackspace (2012). Rackspace Cloud Files. http://www.rackspace.com/cloud/public/files/.

SMPTE (2011). S377m-2011 Material Exchange Format.

Storage Networking Industry Association (SNIA) (2011). Cloud data management interface, version 1.0.1, september 2012 at http://snia.org/sites/default/files/CDMI_SNIA_Architecture_v1.0.1.pdf.

Velocitydb.com (2012). VelocityDB. http://velocitydb.com/.

Versant (2012). Versant website http:// www.versant.com.

World Wide Web Consortium (2004). RDF Primer, feb. 2004, at http://www.w3c.org/TR/rdf-primer/.

Zygmuntowicz, E. (2010). Redis - remote dictionary server, at http://nosql.mypopescu.com/post/408913109/presentation-redis-remote-dictionary-server-by-ezra.

APPENDIX

The following lists the HTTP request trace of a unit test implementation for the relations functionality. It shows how the relations API works on HTTP request level. The requests perform the following steps: (1) Creation of a test container at tenant "siemens" for placing data objects. (2) Creation of one data object named "ccs_test_object_8". (3) Adding this data object to a set named "ccstest_sets_14". Please note that the header fields "Date", "Transfer-Encoding" and "User-Agent" have been omitted due to space restrictions.

```
>>>>
PUT /CCS/_c/siemens/sietestcontainer HTTP/1.1
X-CDMI-Specification-Version: 1.0
Authorization: Basic bWNqQHNpZW1lbnM6c2VjcmV0
Accept: application/cdmi-container
Content-Type: application/cdmi-container
Host: 10.0.1.101:8080
Connection: keep-alive
<<<<<
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-CDMI-Specification-Version: 1.0
Content-Type: application/cdmi-container
{ "objectName": "sietestcontainer/",
  "children": [],
  "metadata": {},
  "capabilitiesURI": "/cdmi_capabilities/container",
  "completionStatus": "Complete",
  "objectURI": "/sietestcontainer/",
  "parentURI": "/",
  "childrenrange": "0-0" }
>>>>>
PUT /CCS/_c/siemens/sietestcontainer/ccs_test_object_8 HTTP/1.1
Authorization: Basic bWNqQHNpZW1lbnM6c2VjcmV0
X-CDMI-Specification-Version: 1.0
Accept: application/cdmi-object
Content-Type: application/cdmi-object
Host: 10.0.1.101:8080
Connection: keep-alive
Content-Length: 56
{"value":"some data","metadata":{"examplekey":"value"}}
<<<<<
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-CDMI-Specification-Version: 1.0
Content-Type: application/cdmi-object
{ "value": "some data",
  "objectName": "ccs_test_object_8",
  "metadata": {
    "cdmi_owner": "mcj@siemens",
    "mimetype": "text/plain",
    "valuetransferencoding": "utf-8",
    "cdmi_size": 11,
    "cdmi_acl": "[{\u0027aceflags...}]",
    "asdf_": "true"
  },
  "mimetype": "text/plain",
  "capabilitiesURI": "/cdmi_capabilities/dataobject",
  "completionStatus": "Complete",
  "objectURI": "/sietestcontainer/ccs_test_object_8",
  "parentURI": "/sietestcontainer/" }
>>>>>
```

```
PUT /CCS/_c/siemens/sietestcontainer/
   ccstest_sets_14/ccs_test_object_8 HTTP/1.1
X-CDMI-Specification-Version: 1.0
Authorization: Basic bWNqQHNpZW1lbnM6c2VjcmV0
Accept: application/cdmi-relation
Content-Type: application/cdmi-relation
Host: 10.0.1.101:8080
Connection: keep-alive
Content-Length: 14
<<<<<
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-CDMI-Specification-Version: 1.0
Content-Type: application/cdmi-relation
{ "context": "0-0",
  "type": "set",
  "children": [ "ccs_test_object_8" ],
  "childrenRange": "0-0" }
```