HITACHI
Inspire the Next

Create a Relational Distributed Object Store

By Robert Primmer, Scott Nyman, Wayzen Lin

October 2014

Contents

Executive Summary

Data storage alone benefits business. But when we can convert stored data to information, its usefulness increases dramatically. Adding a relational layer to the data mass makes this conversion possible. Frank relation among discrete objects that are sporadically ingested is rare, making the process of synthesizing such relation all the more challenging; but the challenge must be met if we are ever to get the same business value from unstructured data that we already get from structured data. This white paper describes a novel construct, referred to as Relational Distributed Object Store (RDOS), that seeks to solve the twin problems of how to persistently and reliably store petabytes of unstructured data while simultaneously creating and persisting relations among billions of objects.

Introduction

Databases have proved useful and versatile containers for structured data, consisting of relatively simple but well-defined data types, referred to as scalars (mostly numbers and strings). In the 1970s, databases evolved beyond basic storing to providing functions critical to the business by allowing relations among the stored data to be expressed and persisted [(1)]. Businesses need to see data correlations from one part of the business to other parts to find insights and operate more efficiently.

As a simple example, consider operations at a retailer such as Walmart, where a database has 2 tables: one records point-of-sale transactions, the other holds inventory data. At the point of sale, it is useful to have a record of all the items purchased; however, it is far more useful to be able to then relate this back to the inventory system to automatically reorder goods as needed. Adding this relational level, and the automation it enables, lets Walmart eliminate manual shelf stock planning, lower costs, prevent over- and under-stocks, and provide significant efficiency to the business.

The container used to store such transactions is typically a relational database, comprising rows (tuples) and columns of scalar data. A substantial strength of relational databases comes in the form of a standard query and transformation logic, structured query language (SQL), which allows applications outside the creating application to query the data[(2)]. The ability to separate data from the bounds of the creating application amplifies and extends the value the data can provide the business.

Database for Unstructured Data

In contrast to the simple scalar types typical to structured data, unstructured data is comprised of rich and expressive data types (such as Microsoft® PowerPoint® presentations or full-motion video) that do not fit well into the traditional database paradigm.

An object store is essentially a database for unstructured data, composed of 2 parts: a distributed database that holds object references and a distributed file store that stores the user data, referred to as data objects or blobs. The database is typically modeled as a NoSQL “shared nothing” data store for horizontal scaling: replicating and partitioning data over many servers[(3)].

Public cloud examples include key-value stores such as Amazon.com, Inc.’s Dynamo[(4)] and Project Voldemort, which is used by LinkedIn Corporation[(5)]. In the enterprise and service provider sectors, Hitachi Content Platform[(6)] (HCP) provides a distributed object store that typically resides behind a firewall[(7)]. HCP is conceptually similar to the combination of Google’s BigTable[(8)] for storing object references and Google File System (GFS)[(9)] for storing data objects.

Abstraction

Fundamental to object storage is that the detail of the distributed database and underlying file systems are abstracted from both the client applications (users) and system administrators. In the process object stores shift the client model, essentially presenting storage as a service rather than requiring clients to be directly involved in data storage decisions, such as properly balancing directory trees. While some object stores only support a single flat namespace(10), others allow the global namespace to be logically partitioned for greater security, as with a collection of buckets in Amazon S3(11), or namespaces in HCP(12).

Abstraction allows for comparatively naïve users and administrators, as the object store takes care of the detail of *how* and *where* user data is stored, protected, geo-replicated, deduplicated, versioned, garbage collected, and so forth. Abstraction also greatly simplifies application development and deployment, as evidenced by the plethora of start-ups, who are able to quickly get a worldwide service up, running and generating revenue by leveraging a public object storage service in unprecedented timeframes.

Likewise, we see where limited compute platforms, such as smartphones and tablets, are able to have full access to a universe of data despite zero support for a single storage protocol. Such devices have no support for the traditional storage protocols [Fibre Channel, Small Computer System Interface (SCSI), Network File System (NFS), and Common Internet File System (CIFS)] made popular in the last century for local-area-network (LAN) attached disk(13). Instead, both simple and complex data types are served by nothing more than the basic Web Hypertext Transfer Protocol (HTTP).

Contributions

This paper describes a mechanism for adding a *relational layer* on top of the object storage layer, without destroying the simplicity gained through the abstractions for which object stores are noted. Our goal is for users to be able to manage relations between data in a fashion similar to a relational database. However, since unstructured data is intrinsically different from structured data, it is necessary that we provide a substrate for defining and describing such relation. Further, we envision creating a mechanism by which users can query an RDOS in a manner similar to the way they query a relational database today for similar benefit.

Document Organization

The remainder of the document is organized as follows. “Unstructured Data” defines unstructured data and touches on some of the associated challenges this data type brings. Metadata is essential to the challenge of creating a relational layer on top of unstructured data. “Metadata” defines metadata, explains why it is of such value, and identifies its potential to transform unstructured data from an undifferentiated data mass to a highly correlated data set that provides genuine value to the business. “Hitachi Content Platform” describes HCP in its present form and “Relational Distributed Object Store” describes the ideal of extending it to be a relational distributed object store and understanding the business value it provides. “Related Work” and “Conclusions” provide a summary of the topics covered in this paper.

Unstructured Data

The awkward term *unstructured data* is intended to complement the term *structured data,* which commonly refers to data stored in a database of some variety. Logically, unstructured data can be thought of as all data not stored in a database, but most commonly it is used to refer to *files* housed in a hierarchical file system. Compared to scalars arranged by row and column within a database, files can be far more expressive, comprising such varied formats as office documents, digital images, and full-motion video, collectively referred to as rich data types.

Objects

The notion of files is further abstracted to objects. While there is no canonical definition for an object in the context of storage, object store implementations tend to represent an object as the union of a data object, file system metadata and (user-created) custom metadata.

Here *data object* is simply the user data; for example a Microsoft Word document that contains a travel itinerary. The file system metadata may include things such as the name of the file, the time it was created and when it was last updated. Thus far, a basic file system would suffice to house both the data (the Word document) and file system metadata (the file name, time created and so forth.). An object store provides the ability to add a 3rd element, *custom metadata*, to the mix.

Custom metadata allows the user to annotate the base data with arbitrary text, often in the form of key-value pairs. Continuing our example, when storing our travel itinerary in an object store we may want to annotate this document with select key data such as: Year=2013, Department=Sales, Territory=US, Status=Approved.

From this example we see that custom metadata provides a mechanism to make the core file more useful by allowing the user to abbreviate the file with select information that can later be used to logically group documents. Additionally, we can perform a lightweight search against the object store for all travel requests by sales in 2013 that were approved for travel within the U.S. and quickly return a list of all objects that meet these criteria.

This is exactly the type of function we would expect to perform against a sophisticated relational database management system (RDBM). However, instead of being restricted to storing scalars, we gain the full expressiveness possible with rich data file types without sacrificing the ability to convert that data into information through mechanisms such as the ability to perform predicate searches. Better still, searching metadata stored within the object store is a far lighter weight operation than searching the full content of every file stored and attempting to piece together logical groupings ad hoc. The value metadata brings to unstructured data is substantial. In short, it is metadata that allows us to add structure where none would otherwise exist in the unstructured data world. We describe this value more deeply in the “Metadata” section.

Data Growth

Throughout the last 2 decades, a recurring topic in information technology (IT) has been how to manage the exponential increase in the volume of data that must be stored as societies move increasingly to a digital world[14]. In addition, while the growth of data to date has been tremendous, the rate of increase is projected to grow greater still. International Data Corporation (IDC) projects that the digital universe will grow by a factor of 300 — to 40,000 exabytes (EB)! — by 2020, and that enterprises will have “liability or responsibility for 80% of the information” in this digital universe. Further, unstructured data is expected to constitute 90% of the 40,000EB[15].

This staggering growth correlates to a shift from text to rich data types such as digital images and video. To get a sense of *how* such data growth can be rationalized, the capacity required to store a high-definition movie trailer is 20,000 times greater than that required to store a traditional movie review[16]. It is this continued shift to rich data types that fuels such spectacular growth, which shows few signs of subsiding.

Data Containers

Expected rapid data growth begs the question: Where are we to store this mountain of data? While RDBMs are used to hold structured data, file systems and object stores are common containers for unstructured data.

Much of the research and development (R&D) over the last 50 years for file systems has focused on how to improve storage efficiency (such as by optimizing file system overhead), durability and reliability. However, from the perspective of the user interface, the basic paradigm of the hierarchical file system remains largely unchanged since its introduction in the 1960s.

For enterprise-class object stores, a cardinal focus has been on answering the question: How do we achieve extreme scale beyond that provided by traditional file systems? Since their genesis, the target market for object stores has been massive data stores, where the feature of cataloging all objects stored has greater value.

It is comparatively easy to keep track of thousands of files for a few years; it is far more difficult to manage billions of them for decades. The Internet and the falling price of magnetic storage have shifted our expectations of digital storage. We now expect data to live in perpetuity, instead of being periodically culled for usefulness. This new behavior further highlights the need for large-scale data management both from a capacity and file count standpoint.

A common misperception is that object stores are a replacement for file systems; instead, they are an augmentation. The file system is tightly coupled to the operating system and provides a well-established mechanism for organizing files within a hierarchy of directories. By contrast, an object store focuses on changing the presentation layer to the storage consumer through a simplified interface while achieving enormous scale by aggregating many file systems into a single, higher-order grouping.

Figure 1 presents an abstract view of the storage stack on a single node. The function of each superior layer is to aggregate and abstract the layer beneath, permitting greater sophistication and specialization in each layer without increasing complexity to clients of upper layers. The object storage layer creates a distributed storage service for client applications without requiring the clients to manage data distribution.

Metadata

In the past the term *metadata* was not widely known, being understood mainly by technologists, and for good reason: File systems provide metadata in parsimonious form, providing data about a file, such as when it was created and last updated.

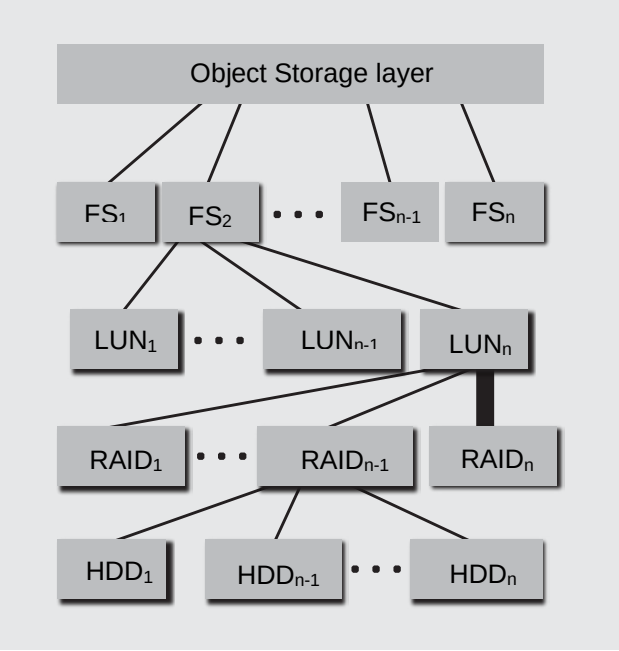

Figure 1. Storage Stack

Object stores by contrast are comparatively lavish in what they allow for metadata, allowing the user to associate any arbitrary text with a data object.

Today, people think nothing of storing their photos online in globally available cloud repositories while annotating the pictures with metadata key-value pairs such as who is in the photo, where it was taken, and at what event. Many also know how to assign metadata tags to logically group items on Facebook and Twitter. In the 21st century, metadata has moved from the exotic to pedestrian.

Value of Metadata

Metadata is the connective tissue that binds objects to one another. Additionally, metadata allows users to apply semantic meaning to otherwise opaque data objects, while providing a means to abbreviate large files with a very small amount of data. In short, it is metadata that allows us to add structure to unstructured data.

How important is metadata? In modern systems if the original designers fail to provide for it, the users do so themselves. Hashtags are a metadata convention among users of the microblogging service Twitter[17], yet when it was first released Twitter had no support for metadata. Instead, hashtags were described by a user in 2007 (in a 140-character Twitter post), and became so popular that Twitter engineers added software support for it. Today, Twitter detects “trending topics” using popular hashtags[18].

Logical Grouping

Absent the ability to create logical groupings, a mass of objects housed in a data store is of limited value. In such a case the data store provides essentially equivalent value as tape: Files are persisted, but there is not much you can do with them.

Logical grouping creates tacit relation among an otherwise indistinct set of independent objects. Since this is a logical operation, the grouping is elastic and does not require expensive disk operations to move files into specific containers to define groupings.

Semantic Meaning

Storing data is one thing, having that same data have genuine meaning to the user is quite another. For example, a photo stored on disk is valuable, but it is the ability to add semantic meaning to the photo (who is in the photo and at what event, for example) that brings life to the data.

Index Efficiency

Indexing the content of very large files consumes substantial capacity to hold the resulting index. Through metadata, it is possible to provide a comparatively small subset of data, effectively creating an abbreviation of the larger file content. The capacity required to index this metadata can be orders of magnitude less than that required to index the full set of data objects, with a commensurate reduction in computes required to create the search index.

Hitachi Content Platform

HCP is a multipurpose distributed object store designed to support large-scale repositories of unstructured data. Physically, HCP is a collection of storage servers (nodes), referred to as a cluster, that use a private back-end network for internode communications and a public front-end network for client communication. A gossip protocol is used to determine active cluster membership. In the event of a temporary failure of any node or service, requests are automatically vectored to active nodes for fulfillment. The system is logically divided into a collection of tenants and namespaces; tenants are the administrative unit, namespaces are the storage unit. Tenants are critical for cloud or service provider deployments where it is important not only to logically separate client data, but

also to divide the administration of each virtual instance of HCP. The system is designed so that no one administrator has dominion over the system as a whole, but only over their assigned tenant. A tenant administrator can create a collection of namespaces that will hold user data.

The value proposition of a multitenant system is largely the same as any shared pool of physical resources, such as storage area networks (SANs): sharing reduces cost by pooling physical resources. In the process, they also expose data on shared storage by unauthorized users and overwrites by multiple clients[19]. To counter this, HCP employs a complex of security measures. These include strictly divided administrative functions, defined user access rights and restrictions, access controls granular to individual objects, and optionally encrypted data, both in flight and on disk.

Client access to the system is via a number of software gateways, distinguished by protocol. Presently, the following protocols are supported: HTTP (REST), S3, CIFS, NFS, Simple Mail Transfer Protocol (SMTP), and Web Distributed Authoring and Versioning (WebDAV). Objects that were ingested using any protocol are immediately accessible through any other supported protocol. These protocols can be used to access the data with a Web browser, HCP client tools, 3rd-party applications, Microsoft Windows® Explorer, or native Windows or Unix tools[12].

HCP provides high availability with strong consistency guarantees. Replication comes in 2 forms: intra- and inter-cluster. Intra-cluster replication is synchronous to the write path to ensure that all data ingested is successfully persisted to disk before returning acknowledgement to the client. Consistent hashing is used to distribute data among the nodes. Within the strictures of assuring replicas are assigned separate fault domains, writes bias toward lightly loaded nodes. Inter-cluster replication is asynchronous to the write path to ensure low write latency while providing geographic dispersion of objects. HCP uses extensible markup language (XML)[20] as the serialization format for persisting custom metadata and XPatH[21] structured queries against that metadata. This allows the system to return specific answers to user queries, rather than a collection of *potential* matches.

HCP employs redundancy at multiple levels: the object reference database, user data (data objects), and system and custom (user) metadata. Likewise, all hardware components are redundant so there is no single point of failure. Data protection centers focus on protecting user data. However, for database systems it is actually the protection of the index that is most important, as a loss of the pointer to the data is tantamount to the loss of the data itself. HCP shares the object reference database among distinct fault domains, both for performance and protection. Further, in the event of catastrophic failure, a scavenging service is employed to reconstruct the database from the constituent metadata persisted to disk as part of the write path.

The data model is one of immutable data objects with mutable metadata. To simulate in-place updates of data objects, HCP supports object versioning, meaning the capability of a namespace to create, store and manage multiple versions of objects within the repository. Capacity efficiency is achieved through a combination of compression and duplicate elimination.

In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, content verification, version control, retention policies, disposition of objects no longer under retention, compression, encryption, failure recovery, replica synchronization, overload handling, state transfer, garbage collection, concurrency and job scheduling, request marshaling, request routing, system monitoring and alarming, and configuration management.

The result is a system that provides data management as a service, referred to as Storage as a Service, in a manner that makes both client development and management of the system simple. The design goal is that neither clients nor system administrators should be aware of the detail of the multiple software services needed to keep a multi-petabyte repository housing billions of objects coherent and responsive to client requests.

Relational Distributed Object Store

The challenges of creating a relational object store are multiple and substantial. The first deals with the most basic question of how do we obtain needed metadata in the first place (“Obtain Metadata”). To solve this we need to begin with a definition of what is needed (“Define”). We can then move on to the problem of how to extract and affine this metadata to the associated data object (“Extraction and Application”), how to create relations (“Create Relations”), how to persist these relations (“Persist Relations”), and finally to put this information to good use through modeling and analytics (“Models and Analytics”).

Obtain Metadata

To date there is relatively little metadata being associated with the data objects in object stores. There are a number of reasons for this.

Application Reluctance: Legacy applications were built before a time when there was a well-defined means for storing and associating metadata with data objects. However, even in the presence of such mechanisms today, a paucity of metadata persists. First, a general application by itself is unlikely to know what constitutes salient metadata for a given file. Second, most commercial applications, recognizing the value of metadata to the customer, are reluctant to give up control of the metadata to be used outside the confines of the application itself. Commercial application providers may not be motivated to put extensive metadata into a data store, as it allows the customer to detach the data from the application, in the process, reducing the stickiness of the application. This customer allowance creates a financial disincentive to enable the user free and ready access to this metadata without the need to involve the creating application.

Further, even if a commercial application provider is properly motivated to do so for the good of the customer, it is hard to determine the right set of data to include in metadata, as each customer is different. Just as with a RDBMs, where individual customers create relations among data sets as appropriate to their own needs, the customer needs to be able to define the metadata that is apropos to their specific environment.

Tools for Association: Few tools exist to enable the systematic association of metadata to data objects. Business agents with the necessary domain expertise for determining what constitutes relevant metadata are unlikely to have the technical expertise required for tool development.

Sophisticated Metadata Stores: Even when modern applications wish to associate meaningful metadata with data objects, few object stores have the sophistication to provide the necessary foundation for good programming practice.

For example, for metadata to be a first-class entity, we must be able to partition the address space so that different users can create, modify and delete their own metadata in isolation. Today, most systems provide a single bucket where all metadata is housed. Changes by one user therefore affect every other user, often in unpredictable ways. To protect user data from such unintended manipulations, all data stores partition the address space for data objects; meaning particular to a function within an organization. For example, one dictionary may contain a grammar common to a particular vertical, such as the field of healthcare, while other dictionaries will be particular to the organization itself, such as code names used within a company. The sum of the individual data dictionaries creates the compendium necessary to inform the extraction and application step.

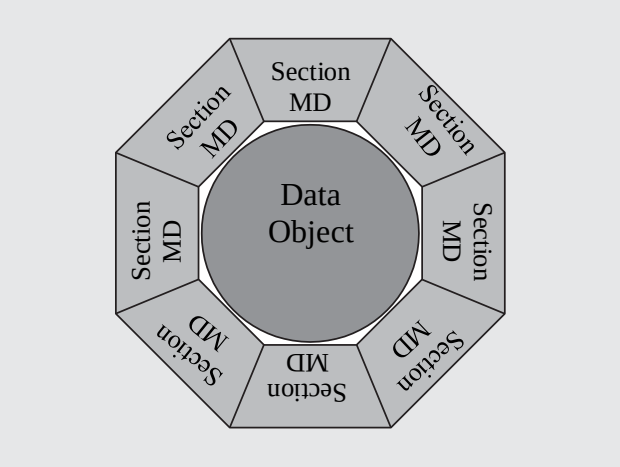

Figure 2. Object With Metadata Partitions

Extraction and Application

The data dictionaries defined in the prior step act as extractors or applicators run across a field of previously ingested data objects or against individual objects during ingestion. When applied as a filter, objects are scanned for relevant key-value pairs within the data object with the results applied as metadata scalars. Of course, such extraction is only possible where the data object is searchable. In cases where it is not, such as for image files, applicators would apply metadata as defined in the data dictionaries to relevant data objects.

Since there can be multiple dictionaries, an object O is run through a pipeline of size p where the rules of each dictionary d are applied in turn. For extraction operations we have $f_1 = \sum_{i=1}^{p} O \cap d_i$, for applicator operations we have $f_2 = \sum_{i=1}^{p} O \cup d_i$. In cases where a particular step in the pipeline has no rules to apply against an object, that step becomes the identity function, that is $O_{input} = O_{output}$. Note that while it is possible that every step in a pipeline will be exclusively of type f_1 or f_2, this is not a requirement; that is, f_1 and f_2 are not mutually exclusive.

Once run through the pipeline, data objects are coupled with relevant metadata and stored as scalars, which can be used for subsequent queries by client applications and to inform relations between objects.

In this model, the extraction or application steps in the pipeline are referred to as a metadata generation module (MGM). As depicted in Figure 3, a data object progresses through a series of MGMs, where an MGM comprises a script language specific to the task.

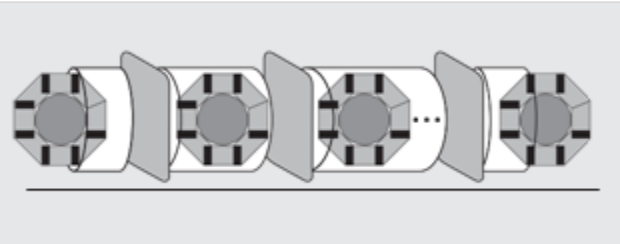

Figure 3. Metadata Generation Pipeline

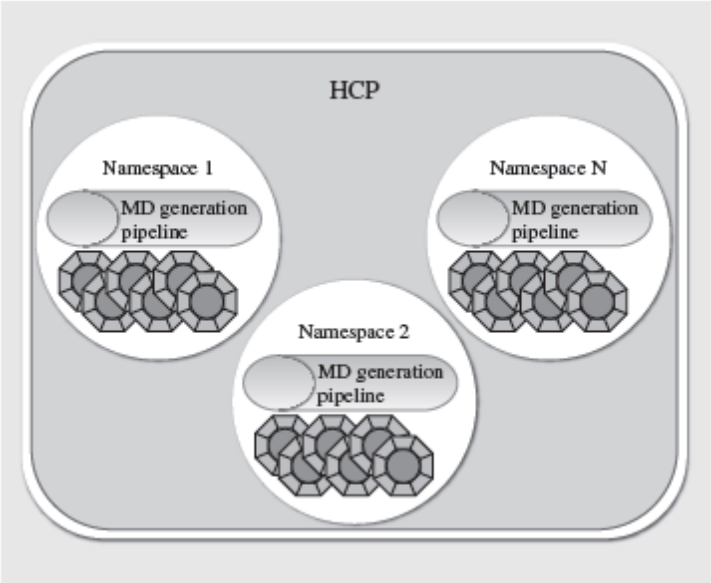

Figure 4. Unique Pipeline per Namespace

At every stage a decision is made whether there is metadata to be applied. While each MGM is independent, the ordering of MGMs can matter; the algorithm applied at any individual MGM may base a decision of whether to add, change or delete a metadata section based on not just the data object, but rather on the union of the data object and metadata accumulated in the pipeline to that point. Nonetheless, fundamental to this design is that an MGM is independent: It allows the flexibility of adding or updating MGMs in existing pipelines.

In Hitachi Content Platform the address space is partitioned into namespaces that essentially act as a virtual instance of the system as a whole. This segregation allows different data management and access policies to be applied to each set of data objects housed within a particular namespace. It is a natural extension that each namespace may have a different set of MGMs to analyze data objects specific to a namespace (see Figure 4). This allows finer-grained decisions to be applied to objects already grouped by user.

Database programmers likely recognize that the MGM construct is conceptually similar to a stored procedure in a RDBMs. A stored procedure is a subroutine stored in the database data dictionary that runs on the database server itself rather than directly on the client. In this model, the MGM acts as a stored procedure, where common code can be run directly on the RDOS cluster rather than on the client.

Note that a client application could choose to perform all the functions of the MGM pipeline directly by reading an object, applying metadata and then rewriting this object back to the RDOS cluster. The downside of this method is that network bandwidth is consumed by the roundtrip for the read/write operations. It can therefore be more efficient to perform this same function on the cluster by running objects through the MGM pipeline. MGM scripts can be either created by the application provider or constructed by the system administrator through a graphical user interface (GUI).

Create Relations

Armed with the metadata from "Extraction and Application," we move to the next step of using this metadata to create relations among objects. For this discussion, we use the example of an auto insurance company servicing a customer claim after a vehicle collision. We have 3 components to the claim: the claim form, digital photographs taken by the appraiser of the damage sustained, and a scan of the police report. These rich data types are good candidates for an object store, as all are unstructured data.

Traditional applications create a database to collect transactions and a schema to associate database tables with one another. However, mobile applications may not wish to require a local database, instead choosing a remote database for this function. In "Database for Unstructured Data," we state that object stores are a database for unstructured data. Here RDOS acts as the database for both the scalar data (in the form of key-value metadata tags) and the data objects (blobs), which consist of any arbitrary file type. To use RDOS in our insurance example, we begin with the claim form where a Claim ID (CID) is assigned (CID=1234) and ingested into RDOS. Subsequently the photos from the appraiser and the scan of the police report are uploaded. For all objects there is a metadata tag set where CID=1234, providing a common key among the objects. At ingest, the client application can choose to add a second metadata field indicating that the images are related to the claim form. As objects are identified by a uniform resource identifier (URI), this translates to: RelTo=URI{Object1}. Through this structure, we have a means of relating the full set of objects that are all associated with the same insurance claim.

This example depends upon the application explicitly creating the relation between the objects by setting the RelTo metadata tag, which is the preferred method for new object ingest. However, the same mechanism described in "Extraction/Application" for adding metadata to objects already stored can be applied here. The final stage of an MGM pipeline can be used to assign the relationship tag to objects in the same manner as all other metadata. The difference in this final MGM script is that it must persist all metadata tags found when enumerating over a set of existing objects. The system administrator can then define how these objects are to be related via a GUI, which in turn creates the needed MGM script.

Persist Relations

Once we have done the hard work of extracting key metadata and presenting mechanisms to the user (both human and application) to define relations, we need to select a means of persisting these relations in a manner that makes sense for the data type. In our model, we have chosen a graph database for this function, as the graph model provides an excellent means of describing relations. Graph database models can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors(2).

Graph databases are described as "whiteboard friendly." Drawing circles and connecting them with lines on a whiteboard is how we visualize graphs. We denote data points as nodes and connect (relate) nodes with links[1] as illustrated in Figure 5. Graph databases, such as NEO4j(22), are well suited to unstructured data, as they are typeless and have no set schema, while providing several advantages for our application.

First, the graph data model is useful when the interconnectivity of data and the ability to discover the relationships between values is important, rather than simple commonality among value sets typical to relational models. Discovering relations is fundamental to our design, so it is important we select a data structure that can readily describe relations with good performance. Unlike join operations in relational databases or map-reduce operations in other databases, graph traversals are in constant time(23). Graph databases make it easy to discover centrality, where we measure individual

[1] Graph theory is a branch of discrete mathematics. Nodes are referred to as *vertices,* and what we call links are referred to as *edges*, where edges connect pairs of distinct vertices. A simple graph (such as we will be using in our descriptions) with V vertices has at most $V(V-1)/2$ edges. We will not be dealing with graphs mathematically other than to note that a graph is defined by its vertices and its edges, not by the way we choose to draw it.

nodes against a full graph (the most famous centrality algorithm is Google PageRank[24]; in “Models and Analytics” we use centrality in an epidemiology example). Finally, graph databases have been shown to perform full-text character searches significantly faster than relational databases[25].

Popular examples of publicly available applications that make use of graph databases include Freebase[26] and the Disease Ontology database[27]. Freebase, created by Metaweb Technologies, Inc., and acquired by Google, Inc., in 2010, is a large community metadata database described as, “an open shared database of the world’s knowledge.” Disease Ontology, created by the University of Maryland school of Medicine, represents a comprehensive knowledge base of 8,043 inherited, developmental and acquired human diseases[28].

Models and Analytics

Once the data is available and programmatically accessible through well-defined application programming interfaces (APIs), we can consider modeling types that users can perform.

Object associations can be definitive or manufactured through a probabilistic generative model to guide inference from incomplete data. The former can be prescribed by the user by linking objects through a GUI or template; the latter through Bayesian inference[30], which provides a rational framework for updating beliefs about latent variables in generative models given observed data[31].

Creating Bayesian models through graphs and predicate logic is more commonly the domain of data scientists than IT users. However, the goal for our system is not to mandate a specific model of relation (definitive or derived), but rather to take care in design to not implicitly restrict the user to a particular model. Our system must provide the flexibility to persist and mutate object relation to meet a variety of user requirements[2].

Our goal is to provide an abstract framework over which schemas can be applied with sufficient flexibility. The framework allows relations to span the prosaic, such as linking monotonically increasing instances of an object (creating a version tree), to the expressive, such as a graph schema where nodes represent variables and directed edges between nodes represent probabilistic causal links.

In an epidemiology study, for example, nodes might represent whether a patient has a cold, a cough, a fever or other condition: The presence or absence of links indicates that colds tend to cause coughing and sinus inflammation but not fever; sinus inflammation tends to cause headache but not fever; and so on[32].

[2] Nate Silver discusses the breadth of problems to which Bayesian reasoning can be applied.

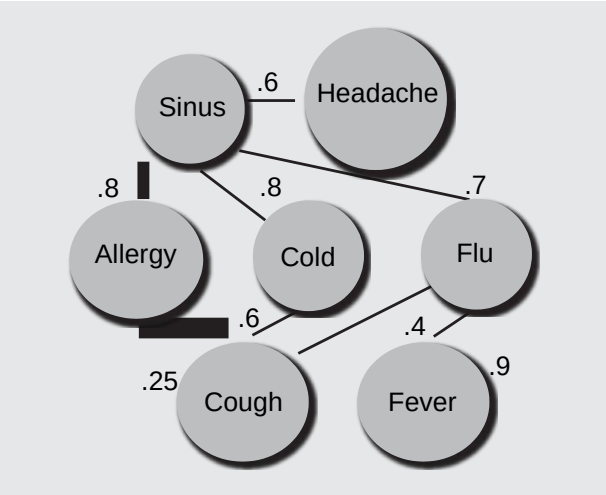

Figure 5. Epidemiology Directed Graph

The probability of a causal relation can be further refined by applying a weighting (0-1.0) to each link. For example, there is a 60% likelihood that a cold will result in a cough. As depicted in Figure 5, graphs provide a ready means for users to interpret data and visualize relationships.

Through the proper application of multipredicate constraints over the field of objects in a domain, the user is able to resolve to classes of objects, such as all objects associated with the progenitor of a disease. For example, a user might want to view all outcomes for patients over 60 in Africa with diminished respiratory function in the presence of the bacteria Legionella, and contrast this to those in the presence of the virus adenovirus to determine whether better outcomes are seen in patients with bacterial versus viral pneumonia. Analytic questions such as this can be compute-intensive when run over the full mass of unstructured data (what we refer to as data objects in “Objects”), but are comparatively lightweight operations when run against the much smaller set of metadata associated with these data objects. Here, the metadata acts an abbreviation of the objects and allows us to create information from an otherwise undistinguished data mass.

Related Work

The NoSQL space has been quite active, with marked similarity among implementations. Distributed systems can be distinguished by how they choose to bias availability versus consistency. Brewer’s consistency, availability and partition (CAP) theorem[34] states that a distributed system can provide for any 2 of the 3 attributes of consistency, availability and partition tolerance. Since networks (particularly wide-area networks or WANs) will always partition (that is, over time there will be some parts of the network that are temporarily unreachable), for distributed systems, this distills to a choice of a design that favors strong consistency or one that is highly available in the face of partitions.

This is the heart of the distinction between SQL and NoSQL systems. SQL systems adhere to atomicity, consistency, isolation and durability (ACID) properties while most NoSQL stores follow basically available, soft state, eventual consistency (BASE) properties. Many NoSQL implementations provide eventual consistency[35], where writes are permitted in the presence of partitions. In such systems, a client can update any replica of an object and all updates to an

object will eventually be applied, but potentially in different orders at different replicas. Thus, the approach creates temporal inconsistencies that must be sorted out by client applications[36].

Amazon's Dynamo[4] is the classic example of the eventually consistent model. Dynamo, a key-value store, is a zero-hop distributed hash table, where each node maintains enough routing information locally to route a request to the appropriate node directly. Dynamo shares a number of characteristics with other distributed object stores used by public cloud vendors that stem from their operational model, which is a closed-loop system where both clients and all services are under the control of a single company. These systems, while used to support public-facing services, can assume that their internal operating environment is non-hostile and therefore have few security requirements such as authentication and authorization. Further, since the NoSQL object stores permit inconsistency, users (client applications) must contain an agreed understanding of how to resolve conflicts during reads. As a general-purpose system with nonspecific clients, HCP can make neither assumption and must design for hostile environments and provide strong consistency guarantees.

Other systems that use an eventual consistency model include Facebook's Cassandra (now Apache) and Google's BigTable. BigTable is a distributed storage system for managing structured data. It maintains a sparse, multidimensional sorted map and allows applications to access their data using multiple attributes[8]. Cassandra has been described as a marriage of Dynamo and BigTable[37]. Yahoo's PNUTS[36] supports Yahoo! Web properties such as Flickr. It uses per-record timeline consistency, where all replicas of a given record apply all updates to the record in the same order. Data objects (blobs) follow the eventually consistent model.

Providing strong consistency guarantees, Microsoft Windows Azure® storage (Was)[38] mixes strong consistency within a local stamp (the analog of a local cluster for HCP) with subsequent replication to remote geographies. This model is the most similar to HCP, which provides strong write guarantees through synchronous replicas locally before asynchronously dispersing replicas geographically. Such systems might be referred to as eventually distributed.

Rather than requiring the object store to deal with collections of file systems, some implementations provide a single (logical) distributed file system, such as the Google File System[9] and CePH[39]. By contrast, HCP uses a confederation of local file systems distributed over a collection nodes to persist data objects. This detail is abstracted from clients in favor of a storage service key-value model, where the key is a URI that indicates a unique object.

Conclusions

In this article we described object stores in general and HCP in particular, discussing how they serve as a data container for unstructured data. We examined the importance of metadata, both to create logical collections of objects, and to provide the basis for establishing relation among objects. We then described an idealized relational distributed object store and considered mechanisms for obtaining and applying metadata to existing objects, and a method of associating and persisting relations among objects, providing a framework for data analytics to be performed against the object store.

References

(1) E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” Commun. ACM 13, 6, pp. 377–387 (Jun. 1970).

(2) R. Angles and C. Gutierrez, “Survey of Graph Database Models,” ACM Comput. Surv. 40, 1, 1:1-1:39 (Feb. 2008).

(3) R. Cattell, “Scalable SQL and NOSQL Data Stores,” SIGMOD Rec. 39, 4, pp. 12–27 (May 2011).

(4) G. DeCandia et al., “Dynamo: Amazon’s Highly Available Key-value Store,” SIGOPS Oper. Syst. Rev. 41, 6, pp. 205–220 (Oct. 2007).

(5) R. Sumbaly et al., “Serving Large-scale Batch Computed Data with Project Voldemort,” Proceedings of the 10th USENIX conference on File and Storage Technologies (Berkeley, CA, USA, 2012), FAST’12, USENIX Association, pp. 18–18.

(6) Hitachi Content Platform, http://www.hds.com/products/file-and-content/content-platform/.

(7) R. Primmer, “Distributed Object Store Principles of Operation,” Tech. rep., Hitachi Data Systems (May 2010).

(8) F. Chang et al., “BigTable: A Distributed Storage System for Structured Data,” ACM Trans. Comput. Syst. 26, 2, 4:1–4:26 (Jun. 2008).

(9) S . Ghemawat, H. Gobioff, and S. T. Leung, “The Google File System,” Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (New York, NY, USA, 2003), SOSP ’03, ACM, pp. 29–43.

(10) R. Primmer, “Efficient Long-term Data Storage Utilizing Object Abstraction and Content Addressing,” Tech. rep., EMC, (Jul. 2003).

(11) J. Varia, “Cloud Architectures,” Tech. rep., Amazon Web Services (Jun. 2008).

(12) M. Ratner, “Hitachi Content Platform: Concepts and Features,” Tech. rep., Hitachi Data Systems (Feb. 2013).

(13) G. Gibson et al., “A Cost-effective, High-bandwidth Storage Architecture,” ACM SIGOPS Operating Systems Review 32, ACM, pp. 92–103 (1998).

(14) A . Adams and N. Mishra, “User Survey Analysis: Key Trends Shaping the Future of Data Center Infrastructure through 2011,” Gartner Market Analysis and Statistics (Oct. 2010).

(15) R. Gants and D. Reinsel, “The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East,” Tech. rep., IDC (Dec. 2012).

(16) R. Primmer, “Structured vs. Unstructured Data,” http://www.robertprimmer.com/blog/structured-vs-unstructured.html (2012).

(17) H. Kwak, C. Lee, and S. Moon, “What is Twitter, a Social Network or a News Media?” Proceedings of the 19th International Conference on World Wide Web (New York, NY, USA, 2010), WWW ’10, ACM, pp. 591–600.

(18) A . Badia and D. Lemire, “A Call to Arms: Revisiting Database Design,” SIGMOD Rec. 40, 3, pp. 61–69 (Nov. 2011).

(19) H. Yoshida, “LUN Security Considerations for Storage Area Networks,” Hitachi Data Systems PaperXP 2185193, pp. 1–7 (1999).

(20) Extensible Markup Language (XML), http://www.w3.org/XML.

(21) XML Path Language (XPath) Version 1.0, http://www.w3.org/TR/xpath.

(22) J. Webber, “A Programmatic Introduction to Neo4j,” Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (New York, NY, USA, 2012), SPLASH ’12, ACM, pp. 217–218.

(23) E. Redmond and J. Wilson, “Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement,” O'Reilly and Associate Series, O’Reilly Vlg. GmbH & Company (2012).

(24) L. Page et al., “The Pagerank Citation Ranking: Bringing Order to the Web,” Technical Report 1999-66, Stanford InfoLab (Nov. 1999). Previous Number = SIDL-WP-1999-0120.

(25) C. Vicknair et al., “A Comparison of a Graph Database and a Relational Database: a Data Provenance Perspective,” Proceedings of the 48th Annual Southeast Regional Conference (New York, NY, USA, 2010), ACM SE ’10, ACM, pp. 42:1-42:6.

(26) Freebase, http://www.freebase.com/.

(27) Disease Ontology, http://disease-ontology.org/.

(28) L. M. Schriml et al., “Disease Ontology: a Backbone for Disease Semantic Integration,” Nucleic Acids Research 40, D1 (2012), D940-D946.

(29) R. Sedgewick, “Algorithms in C++ - Part 5: Graph Algorithms (3 .ed.),” Addison-Wesley-Longman (2002).

(30) J. Pearl, “Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference,” Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, (1988).

(31) D. J. C. MacKay, “Information Theory, Inference & Learning Algorithms,” Cambridge University Press, New York, NY, USA, (2002).

(32) J. Tenenbaum et al., “How to Grow a Mind: Statistics, Structure, and Abstraction,” science 331, 6022, pp. 1279–1285 (2011).

(33) N. Silver, “The Signal and the Noise: Why So Many Predictions Fail—but Some Don’t,” Penguin Group US, (2012).

(34) S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services,” SIGACT News 33, 2, pp. 51–59 (Jun. 2002).

(35) W. Vogels, “Eventually Consistent,” Queue 6, 6, pp. 14–19 (Oct. 2008).

(36) B. F. Cooper et al., “PNUTS: Yahoo!’s Hosted Data Serving Platform,” Proc. VLDB Endow. 1, 2, pp. 1277–1288 (Aug. 2008).

(37) A. Lakshman and P. Malik, “Cassandra: a Decentralized Structured Storage System,” SIGOPS Oper. Syst. Rev. 44, 2, pp. 35–40 (Apr. 2010).

(38) C. Huang et al., “Erasure Coding in Windows Azure Storage,” Proceedings of the 2012 USENIX Conference on Annual Technical Conference (Berkeley, CA, USA, 2012), USENIX AT C’12, USENIX Association, pp. 2–2.

(39) S . A. Weil et al., “Ceph: a Scalable, High-performance Distributed File System,” Proceedings of the 7th Symposium on Operating Systems Design and Implementation (Berkeley, CA, USA, 2006), OSDI ’06, USENIX Association, pp. 307–320.

(40) C. P. Wright et al., “Extending Acid Semantics to the File System,” Trans. Storage 3, 2 (Jun. 2007).

Hitachi Data Systems

Corporate Headquarters
2845 Lafayette Street
Santa Clara, CA 95050-2639 USA
www.HDS.com **community.HDS.com**

Regional Contact Information
Americas: +1 866 374 5822 or info@hds.com
Europe, Middle East and Africa: +44 (0) 1753 618000 or info.emea@hds.com
Asia Pacific: +852 3189 7900 or hds.marketing.apac@hds.com

WP-495-B J. Lundberg September 2016