Creating a relational distributed object store

Robert Primmer, Scott Nyman, Wayzen Lin
Hitach Data Systems
{bob.primmer,scott.nyman,wayzen.lin}@hds.com

June 2013

Abstract

In and of itself, data storage has apparent business utility. But when we can convert data to information, the utility of stored data increases dramatically. It is the layering of relation atop the data mass that is the engine for such conversion. Frank relation amongst discrete objects 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 see an equivalent business value for unstructured data as we already have with structured data. This paper describes a novel construct, referred to as a 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 amongst billions of objects.

1 Introduction

Databases have proven to be a useful and versatile container for housing structured data, consisting of relatively simple but well-defined data types, referred to as *scalars* (mostly numbers and strings). In the 1970's databases evolved beyond a basic storage container to provide functions critical to the business by allowing relations amongst the stored data to be expressed and persisted [6]. This is fundamental to how business operates—where data from one part of the business must be correlated to other parts of the business if insights and efficiencies are to be realized.

As a simple example, consider operations at a retailer such as Walmart where a database has two tables: one records point-of-sale transactions, the other holds inventory data. At the point-of-sale it's useful to have a record of all the items purchased, however it's far more useful to be able to then relate this back to the inventory system to automatically reorder goods as needed. This relation, and the automation it enabled, let Walmart eliminate manual shelf stock planning, lowered cost, and prevented over- and under-stocks. This basic example shows where adding relation provides significant efficiency to the business.

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

1.1 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 (e.g. 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 two 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 [4].

Public cloud examples include key-value stores such as Amazon's Dynamo [8] and Project Voldemort used by LinkedIn [27]. In the enterprise and service provider sectors, the Hitachi Content Platform[1] (HCP) provides a distributed object store that typically resides behind a firewall [20]. HCP is conceptually similar to the combination of Google's Bigtable

---

[1]http://www.hds.com/products/file-and-content/content-platform

1

[5] for storing object references and Google File System (GFS) [10] for storing data objects.

1.2 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 [19], others allow the global namespace to be logically partitioned for greater security, e.g. with a collection of *buckets* in Amazon S3 [29], or *namespaces* in HCP [22].

Such 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, de-duplicated, versioned, garbage collected, and so forth. This grossly simplifies application development and deployment, as evidenced by the plethora of start-ups who are able in short order to get a worldwide service up, running, and generating revenue by leveraging a public object storage service—something never seen before in history.

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 traditional storage protocols (FibreChannel, SCSI, NFS, CIFS) made popular in the last century for LAN-attached disk [11]. Instead both simple and complex data types are served by nothing more than the basic web protocol HTTP.

1.3 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 similar fashion to a relational database. However, since unstructured data is intrinsically different from structured data, it's 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.

1.4 Document Organization

The remainder of the document is organized as follows. Section 2 defines unstructured data and touches on some of the associated challenges this datatype brings. Metadata is essential to the challenge of creating a relational layer on top of unstructured data. Section 3 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. Section 4 describes HCP in its present form and section 5 describes the ideal of extending it to be a relational distributed object store and the business value it provides. Related work is described in section 6 and section 7 concludes with a summary of the topics covered in this paper.

2 Unstructured Data

The awkward term *unstructured data* is intended to complement the term *structured data*—which commonly refers to data stored in a database (DB) of some variety. Logically, unstructured data can be thought of as all data not stored in a DB, 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 DB, 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*.

2.1 Objects

The notion of files is further abstracted to *objects*. While there isn't a canonical definition specifying precisely what constitutes an object in the context of storage, object store implementations tend to represent *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 filename, time created, etc.). An object store provides the ability to add a third element, *custom metadata*, to the mix.

2

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 US and quickly return a list of all objects that meet these criteria.

This is exactly the type of function we'd expect to perform against a sophisticated RDBMS. 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's metadata that allows us to add structure where none would otherwise exist in the unstructured data world. In §3 we describe this value more deeply.

2.2   Data Growth

Throughout the last two decades a recurring topic in IT is how to manage the exponential increase in the volume of data that must be stored as societies move increasingly to a digital world [1]. What's more, while the growth of data to date has been tremendous, the rate of increase is projected to grow greater still. IDC projects that the *Digital Universe* will grow by a factor of 300—to 40,000 exabytes!—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 40,000 exabytes [9].

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, in [21] we see where the capacity required to store a high definition movie trailer is 20,000 times greater than that required to store a traditional movie review. It's this continued shift to rich data types that fuels such spectacular growth; and this growth shows few signs of subsiding.

2.3   Data Containers

Such 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 R&D over the last 50 years for file systems has focused on how to improve storage efficiency (e.g. optimizing file system overhead), durability, and reliability. However, from the perspective of 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's comparatively easy to keep track of thousands of files for a few years; it's 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* to client applications without requiring the clients to manage data distribution.
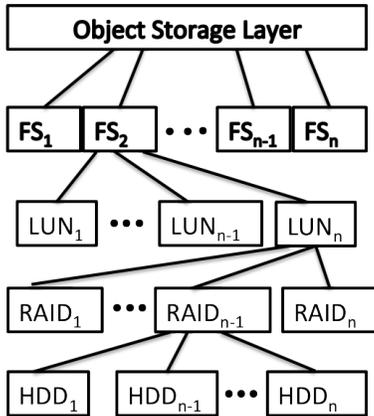
Figure 1: Storage Stack

3 Metadata

In the past the term *metadata* was not widely known, understood mainly by technologist, 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. 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, grandmothers 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. Young children 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.

3.1 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 amongst users of the microblogging service Twitter [14], 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 the engineers at Twitter added software support for it. Today, Twitter detects "trending topics" using popular hashtags [3].

3.1.1 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's not much you can do with them.

Logical grouping creates tacit relation amongst 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.

3.1.2 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's the ability to add semantic meaning to these photos (e.g., who is in the photograph at what event) that brings life to the data.

3.1.3 Index Efficiency

Indexing the content of very large files consumes substantial capacity to hold the resulting index. Through metadata it's 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.

4 HCP

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 inter-node communications and a public front-end network for client communication. A gossip protocol is used determine active cluster membership. In the event of temporal 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 / service provider deployments where it's 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 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 [35]. To counter this HCP employs a complex of security measures. These include strictly dividing administrative functions, defined user access rights and restrictions, access controls granular to individual objects, and all data is optionally encrypted, 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, SMTP and Web-DAV. 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, third-party applications, Windows Explorer, or native Windows or Unix tools [22].

HCP provides high availability with strong consistency guarantees. Replication comes in two 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 XML[2] as the serialization format for persisting custom metadata and XPATH[3] structured queries against that metadata. This allows the system to return *specific* answers to user queries, rather than a collection of *potential* matches.

---

[2] http://www.w3.org/XML
[3] http://www.w3.org/TR/xpath

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. Commonly the focus for data protection centers on protecting user data. However, for database systems it's actually the protection of the index that's most important, as a loss of the pointer to the data is tantamount to the loss of the data itself. HCP shards the object reference database amongst 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, i.e. 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.

5 RDOS

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 (§5.1). To solve this we need to begin with a definition of what is needed (§5.2). We can then move on to the problem of how to extract and affine this metadata to the associated data object

(§5.3), how to create relations (§5.4), how to persist these relations (§5.5), and finally to put this information to good use through modeling and analytics (§5.6).

5.1 Obtaining 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 creates a financial disincentive to allow 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's hard to determine what is the right set of data to include in metadata as each customer is different. Just as with a RDBMS, where individual customers create relations amongst 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 exists 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 it's necessary that we are 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; few provide equivalent functionality for metadata. For these reasons HCP allows metadata partitions for each data abject. Figure 2 illustrates this concept with a data object that has eight distinct metadata partitions. In §5.3 we prescribe a mechanism for users to populate these metadata partitions for both new data ingest, as well as for existing objects already stored.
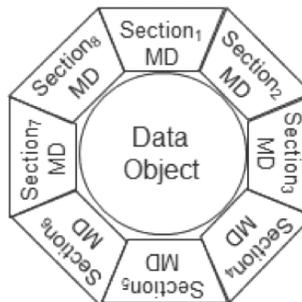

Figure 2: Object with Metadata Partitions

Finally, it's critical that an object store can scale—not just in its ability to store petabytes of data, but to be able to keep track of billions of discrete objects and associated metadata. The former is comparatively easy, the latter is far more difficult. With HCP, each node can independently manage in excess of 800 million data objects and associated metadata, a full cluster can manage 64 billion.

5.2 Defining

The first step is to define what metadata should be extracted or applied to data objects. Search engines can crack many file types and extract not only data from the core file, but also in some cases metadata embedded in the file's header (e.g. in the case of DICOM images). However, standard data is is made far more useful by applying *semantic meaning* to the content. This step is difficult for a general search engine to do as, by its very nature, semantic meaning is often particular to a user or organization and therefore doesn't lend itself to a generic solution.

Creating an ontology, where semantic meaning can be mapped to data, is referred to as a *Data Dictionary*. Multiple such dictionaries can be created, each with meaning peculiar to a function within an organization. For example, one dictionary may contain a grammar common to a particular vertical, such as

6

the field of healthcare, while other dictionaries will be particular to the organization itself, such as codenames used within a company. The sum of the individual data dictionaries creates the compendium necessary to inform the extraction and application step.

5.3 Extraction / 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's possible that every step in a pipeline will be exclusively of type f_1 or f_2, this is not 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.
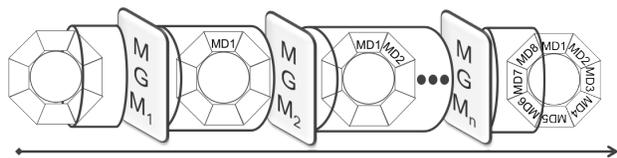

Figure 3: Metadata Generation Pipeline

In this model the extraction or application steps in the pipeline are referred to as a *metadata generation module*, or 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. 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 as the algorithm applied at any individual MGM may base

a decision of whether to add, change or delete a metadata section based not only the data object itself, but rather 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, to allow the flexibility of adding or updating MGMs in existing pipelines.

In HCP 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 (Figure 4). This allows finer-grain decisions to applied to objects already grouped by user.
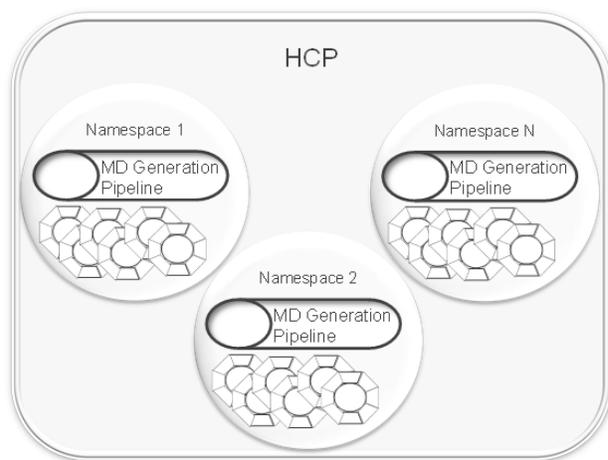

Figure 4: Unique Pipeline per Namespace

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 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 cre-

ated either by the application provider or constructed by the system administrator through a GUI.

5.4 Creating Relations

Armed with the metadata from §5.3 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 three 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 create 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 §1.1 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 amongst 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 URI, this translates to: `RelTo=URI{Object1}`. Through this structure we have a means of relating the full set of objects which 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 §5.3 for adding metadata to objects already stored can be applied here. The final stage of a 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.

5.5 Persisting Relations

Once we've 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 relation. 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*[4], as illustrated in Figure 5 (p. 9). Graph databases, such as NEO4j [32], 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 simply commonality *among value sets* typical to relational models. Discovering relations is fundamental to our design, so its 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 nodes against a full graph. (The most famous centrality algorithm is Google PageRank [17]. In §5.6 we use centrality in an epidemiology example.) Finally, graph databases have been shown to perform full-text character searches significantly faster than relational databases [30].

Popular examples of publicly available applications that make use of graph databases include Freebase[5] and the Disease Ontology database[6]. Freebase, cre-

---

[4]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'll be using in our descriptions) with V vertices has at most $V(V-1)/2$ edges. We won't 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 [25].

[5]http://www.freebase.com
[6]http://disease-ontology.org

ated by Metaweb and acquired by Google 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 8043 inherited, developmental and acquired human diseases [24].

5.6 Models and Analytics

Once the data is available and programmatically accessible through well-defined APIs, modeling types that users can perform can be considered.

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 [18], which provides a rational framework for updating beliefs about latent variables in generative models given observed data [16].

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 isn't 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[7].

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

For example, in an epidemiology study nodes might represent whether a patient has a cold, a cough, a fever or other conditions, and 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 [28]. The probability of a causal relation can be further refined by applying a weighting (0 - 1.0) to each link; e.g. there's 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.

---

[7]Nate Silver discusses the breadth of problems to which Bayesian reasoning can be applied in [26].
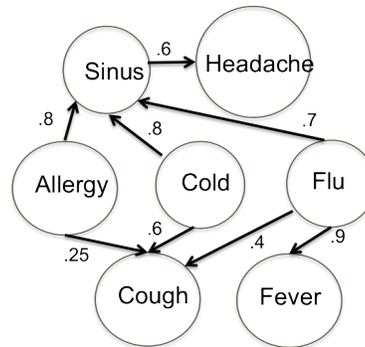

Figure 5: Epidemiology Directed Graph

Through the proper application of multi-predicate 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.

Such analytic questions can be compute intensive when run over the full mass of unstructured data (what we refer to as *data objects* in §2.1), 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.

6 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 CAP theorem [12] states that a distributed system can provide for any two of the three attributes of *Consistency, Availability* and *Partition tolerance.* Since networks (particularly WANs) will always partition (i.e., 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 ACID properties while most NoSQL stores follow BASE

properties[8]. Many NoSQL implementations provide *eventual consistency* [31], 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 creating temporal inconsistencies that must be sorted out by client applications [7].

Amazon's Dynamo [8] 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 non-specific 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, multi-dimensional sorted map and allows applications to access their data using multiple attributes [5]. Cassandra has been described as a marriage of Dynamo and Bigtable [15]. Yahoo's PNUTS [7] 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's Windows Azure Storage (WAS) [13] 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 [10] and CEPH [33]. 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.

7 Summary

In this paper we described object stores in general and HCP in particular, describing how they serve as a data container for unstructured data. We discussed 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 described mechanisms for obtaining and applying metadata to existing objects, and a method of associating and persisting relations amongst objects, providing a framework for data analytics to be performed against the object store.

References

[1] ADAMS, A., AND MISHRA, N. User survey analysis: Key trends shaping the future of data center infrastructure through 2011. *Gartner Market Analysis and Statistics* (Oct. 2010).

[2] ANGLES, R., AND GUTIERREZ, C. Survey of graph database models. *ACM Comput. Surv. 40*, 1 (Feb. 2008), 1:1–1:39.

[3] BADIA, A., AND LEMIRE, D. A call to arms: revisiting database design. *SIGMOD Rec. 40*, 3 (Nov. 2011), 61–69.

[4] CATTELL, R. Scalable sql and nosql data stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[6] CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.

[7] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R.

---

[8]ACID stands for Atomicity, Consistency, Isolation and Durability; BASE stands for Basically Available, Soft state, Eventual consistency. Descriptions of ACID and BASE can be found in [34] and [4] respectively.

Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev. 41*, 6 (Oct. 2007), 205–220.

[9] Gants, R., and Reinsel, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. Tech. rep., IDC, Dec. 2012.

[10] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[11] Gibson, G., Nagle, D., Amiri, K., Butler, J., Chang, F., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *ACM SIGOPS Operating Systems Review* (1998), vol. 32, ACM, pp. 92–103.

[12] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (June 2002), 51–59.

[13] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 2–2.

[14] Kwak, H., Lee, C., Park, H., and Moon, S. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 591–600.

[15] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[16] MacKay, D. J. C. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, New York, NY, USA, 2002.

[17] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[18] Pearl, J. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[19] Primmer, R. Efficient long-term data storage utilizing object abstraction and content addressing. Tech. rep., EMC, July 2003.

[20] Primmer, R. Distributed object store principles of operation. Tech. rep., Hitachi Data Systems, May 2010.

[21] Primmer, R. Structured vs. unstructured data. http://www.robertprimmer.com/blog/structured-vs-unstructured.html, 2012.

[22] Ratner, M. Hitachi content platform: Concepts and features. Tech. rep., Hitachi Data Systems, Feb 2013.

[23] Redmond, E., and Wilson, J. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* Oreilly and Associate Series. O'Reilly Vlg. GmbH & Company, 2012.

[24] Schriml, L. M., Arze, C., Nadendla, S., Chang, Y.-W. W., Mazaitis, M., Felix, V., Feng, G., and Kibbe, W. A. Disease ontology: a backbone for disease semantic integration. *Nucleic Acids Research 40*, D1 (2012), D940–D946.

[25] Sedgewick, R. *Algorithms in C++ - part 5: graph algorithms (3 .ed.).* Addison-Wesley-Longman, 2002.

[26] Silver, N. *The Signal and the Noise: Why So Many Predictions Fail—but Some Don't.* Penguin Group US, 2012.

[27] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 18–18.

[28] Tenenbaum, J., Kemp, C., Griffiths, T., and Goodman, N. How to grow a mind: Statistics, structure, and abstraction. *science 331*, 6022 (2011), 1279–1285.

[29] Varia, J. Cloud architectures. Tech. rep., Amazon Web Services, June 2008.

[30] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference* (New York, NY, USA, 2010), ACM SE '10, ACM, pp. 42:1–42:6.

[31] Vogels, W. Eventually consistent. *Queue 6*, 6 (Oct. 2008), 14–19.

[32] Webber, J. A programmatic introduction to neo4j. In *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.

[33] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[34] WRIGHT, C. P., SPILLANE, R., SIVATHANU, G., AND
     ZADOK, E. Extending acid semantics to the file sys-
     tem. *Trans. Storage 3*, 2 (June 2007).

[35] YOSHIDA, H. Lun security considerations for stor-
     age area networks. *Hitachi Data Systems PaperXP
     2185193* (1999), 1–7.