The Universal Data Cube

Curran Kelleher

Contents

Part I

The Universal Data Cube

Data visualization and analysis tools often lack explicit support for hierarchical data cubes, their metadata, and dynamic merging of comparable data from multiple sources. The Universal Data Cube is a vision for a world wide web in which richly annotated hierarchical data cubes are first class citizens and rich web-based data visualization and analysis tools are commonplace. The Universal Data Cube (UDC) vision requires many distinct components in order to function and flourish: the UDC Model provides the conceptual fountation of the system; the UDC Ontology is the concrete realization of the UDC Metadata Model using Semantic Web technologies; the UDC Core Library is a software specification of the UDC Model. These three parts form the basis of an open infrastructure upon which tools for data publishing, data navigation, interactive visualization and analysis can be built.

Chapter 1

Conceptual Model

The Universal Data Cube theoretical model (UDC model) is a conceptual framework which characterizes distributed hierarchical data cubes in terms of their content, structure, and metadata. While this model is theoretical, it is designed specifically to solve many concrete and ubiquitous problems in data publishing, navigation, interactive visualization and analysis:

- No open standard for publishing and consuming hierarchical data cubes with rich metadata exists.
- The metadata necessary to determine whether two data sources provide comparable data is often not machine readable, therefore merging of data sources requires extensive manual effort.
- Data sources using different coding schemes for identifying records require manual work (creation of code translation tables) to merge and compare.
- Data sources representing numeric values with the same meaning using different units require manual work (creation of unit conversion functions) to merge and compare.
- The data infrastructure for many visualization tools must still often be developed from scratch, and is domain or task specific.
- Hierarchical and tabular data are widely considered findamentally different kinds of data, and tools are often developed for only one or the other.

- The work required to create a tool in which users can navigate categorical or regional hierarchies is repeated for every such tool created.

All these problems are solved effectively by the UDC model.

A data cube is an a form of data abstraction which is comprised of *dimensions* (containing objects) and *measures* (representing properties). A *hierarchical data cube* is a data cube whose dimensions may be hierarchical. The UDC model is an extention of the hierarchical data cube concept which aims to provide a comprehensive framework for describing and using data cubes. We first introduce the structural terms used in the UDC Model, partitioned into those relating to dimensions and those relating to measures. We then then discuss how these terms can be composed to form the conceptual framework for a distributed data system potentially containing all possible hierarchical data cubes, namely the *universal data cube*. We conclude with a discussion of how this structure can be practically used for data publishing, navigation, interactive visualization and analysis.

1.1 Dimensions

A *record* represents a region of time, a region of space, an object category or an individual object. Example records include the year "1990" (a region of time), the country "USA" (a region of space), the industry sector "Mining" (an object category), and the iris flower with id "25" (an individual object). Records can be hierarchical. Each record may have a *parent record*. For example, the parent record of the US state "Massachusetts" may be the country "USA". For a given record, the records which have it as a parent are considered its *child records*.

Each record can have more than one parent. Therefore the structure of record relationships is not limited to a hierarchy (tree), but in general is a *topology* (graph). A record a is considered a parent of another record b if and only if the region, category or individual represented by b is fully contained within that represented by a. Many *record hierarchies* (containment trees) can be expressed within a single *record topology* (containment graph).

A *record product* can be defined for regional or categorical records, which represents a composite region. For example, the product of the country "USA" and the year "1990" defines simultaneously a region of time and space, namely "The USA in the year 1990".

A *level* represents a class of records which are on the same level in a record hierarchy. Example levels include "Year", "Country", "Industry Sector" and "Iris". Levels can be hierarchical. Each level may have a *parent level*. For example, the parent record of the level "US state" may be the level "Country". For a given level, the levels which have it as a parent are considered its *child levels*. A level tree represents a record topology. A path from the root to a leaf of a level tree (or any subpath thereof) represents a record hierarchy.

A *dimension* represents a set of levels which could all potentially belong to the same level tree, and the record topology represented by these levels. Example dimensions include "Time", "Space", "Industry" and "Iris Category". All records are contained within levels, and all levels are contained within dimensions.

1.2 Measures

A *quantity* is a kind of numeric property. Example quantities include Currency, Quantity of People, Mass, and Speed. A *unit* is a concrete realization of a quantity. Example units include US Dollars, Thousands of People, Kilograms, and Kilometers per Hour. An *aggregation operator* defines a method of aggregating numeric values. Example aggregation operators include "Sum" and "Average". A *measure* is a numeric property of records or products thereof, defined by a quantity and an aggregation operator. Here are some example measures, shown in the form: "measure name", "quantity", "aggregation operator":

- "Average Income", "Currency", "Average"
- "Population", "Quantity of People", "Sum"
- "Employment", "Quantity of People", "Sum"
- "Average Speed Limit", "Speed", "Average"

1.3 Data

A *data cube* in general is a mapping from record products to measure values. For example, a data cube of the US Census data would contain a mapping

from the record product "The USA in the year 1990" to a value for the measure "Population" in a specific unit.

A *dimension instance* is a specific subset of records from one level of a dimension. A *measure instance* is a concrete realization of a measure annotated with its unit, in other words a $(measure, unit)$ pair. The structure of a data cube is defined by a set of dimension instances and a set of measure instances. The *cells* of a data cube are defined by all possible record products in which one record is taken from each of the data cube's dimension instances. The content of data cube is defined by the mapping from its cells to numeric values for each of its measure instances.

A *dataset* is a collection of data cubes. A dataset whose data cubes all contain the same set of measures instances and contain levels representative of the same set of dimensions can be interpreted as a data cube in which each dimension is hierarchical. Such a data set is called a *hierarchical data cube*.

1.4 Properties

The UDC model can be cleanly divided into realms of *knowledge* and *data*. Knowledge in this context refers to the terminology potentially used across data sets: records, levels, dimensions, quantities, units, and measures. Data in this context refers to actual data set contents and their descriptors: dimension instances, measure instances, data cubes and data sets.

1.5 Auxiliary Data

Code tables are mappings from strings of a specific coding scheme to record URIs. *Unit conversion tables* are mappings from unit pairs to conversion factors.

1.6 Operations

The knowledge realm of the UDC model can be represented as a semantic graph (a graph with labeled edges). Any semantic graph query on this knowledge base is possible.

Data cubes can be projected (queried) by selecting a subset of their records and measures. We use the notion of a *record selection* for describing subsets of records used when specifying a projection. A record selection in a given dimension instance can be defined by a single record representing the root of a record subtree. The records at the level represented by the given dimension instance are selected.

Data cubes are comparable when their set of dimension instances represent the same set of levels, and their set of measure instances represent the same set of measures. Comparable data cubes can be merged by taking the union of records in their respective dimension instances, and harmonizing (converting) units of comparable measures when necessary (using unit conversion tables).

Chapter 2

Ontology

Document content goes here

Chapter 3

Core Library

3.1 Introduction

The UDC Core Library provides two distinct components: the metadata framework, a specification enabling the UDC metadata system to be realized in memory; and the data cube API, a specification for data cube implementations. The aim of the UDC Core Library is to provide a minimalistic yet comprehensive API specification (*not* an implementation) which provides a common language for all components of the UDC System.

3.2 The Metadata Framework

The UDC Core Library specifies a system in which knowledge in the Universal Data Cube metadata system can be partially reconstructed in memory. Interfaces were created for each UDC Ontology class. The purpose of this is to enable semantic web Resources published using the UDC Ontology to be dynamically loaded into memory as class instances.

A key contract which must be adhered to is that each resource be represented by exactly one in-memory class instance. Implementations of the UDC Core API should enforce this contract by using the factory pattern, always consulting a global index mapping URIs to class instances before creating new ones. This contract affords efficient implementation of data cubes whether they reside in a database or in memory, and convenient usage of their resulting class instances by clients of UDC Core.

Uniqueness of resource instances is preserved by maintaining an index

mapping global URIs to local class instances, and enforcing (through use of the Factory design pattern) its use whenever new resources are consumed.

In addition to semantic web resources being mapped to class instances, the properties of these semantic web resources are mapped to class member variables. This enables very efficient implementations of metadata queries relying on graph traversal such as "What are all the counties of Texas?" or "What are all the months from 1990 to 1995?". Traversing the JVM object graph directly in order to answer these queries is likely much more efficient than relying on a SPARQL implementation. In addition, providing classes, properties, and operations at the programming language level makes code much cleaner than it would be if one had to instead always interact with an RDF model via SPARQL and other tools.

3.3 The Data Cube Interface

In addition to the classes and properties found in the UDC Ontology, interfaces for defining data cube implementations are also part of the UDC Core Library.

Chapter 4

SQL Library

4.1 Introduction

The UDC SQL component provides two concrete data cube implementations which use a SQL database as a data store: one writable data cube implementation based on the star schema concept, and one read-only data cube implementation for exposing existing databases as data cubes without modification.

4.2 Star Schema Based Implementation

A star schema is a common and efficient way of storing data cubes in relational databases. In a star schema, there is a central table which has one row for each data cube cell, containing its address and its measure values. This central table is called the *fact table*. In the fact table, data cube cells are addressed by combinations of record keys. The record keys are resolvable into record descriptors via *dimension tables*. Each dimension of the data cube has its own dimension table which is just a mapping between record keys and record descriptors.

4.3 Database Wrapper Implementation

Publishing data in existing databases without modification is an essential requirement of the UDC System. To accomplish this, a data cube implemen-

tation which wraps existing databases was created.

Chapter 5

Client Framework

The UDC client framework is comprised of the following components:

- Application State Historian
- UDC Signal Processing Framework
- Interactive Graphics Library
- UDC Visualization Tools

5.1 Application State Historian

The purpose of the Application State Historian (ASH) framework is to provide a dynamic application state model supporting session history navigation and synchronous collaboration. The central data structure of ASH is called the *session model*, and is a simple recursive data structure similar to the XML DOM: nodes have type, properties and child nodes. Applications using ASH use this structure as their central application model. An instance of this model is called a *session state*. Each action a user performs is encapsulated in a *session state transition*. Every session state transition has a performable inverse. The graph induced by session states and state transitions is called the *session graph*. ASH provides the functionality of navigating between any two nodes in a session graph. ASH also provides the functionality of keeping many ASH session states synchronized across the internet.

ASH is based on a plugin architecture which contains a mapping between node type names and classes. ASH is a runtime component manager, and

components are created by ASH plugins, which implement the factory pattern for nodes of a specific type. When a new node of a given type is added to the session model, an instance of the class assigned to that node type is created.

An ASH model as a whole is a tree data structure which has the additional property that each node has a unique numeric id, the ordering of which reflects the order (in time at runtime) in which the objects (nodes in the object dependency graph) were created. This allows ASH objects to refer to other with guaranteed referential integrity at runtime. Referential integrity of the live object graph in at runtime is guaranteed because an ascending sort of the ASH objects creation time (i.e. sort by id) is equivalent to a topological sort on the ASH object dependency graph.

Because of these properties, a live ASH object dependency graph can be persisted in a stack data structure in which elements of the stack can refer to other elements whose id is less than its own. With this new view, node creation and deletion operations can be thought of as push and pop operations on this stack. This gives the ASH model two fundamental global level operations: create node and delete node. Create node returns a number which refers to that node.

There are three simultaneous data structures represented in an live ASH model: the object containment tree, the runtime object dependency graph, and the object stack. In a proxy ASH model used by collaboration servers, only the object containment tree and object stack are present.

The ASH session graph represents a state transition graph between session states. A live ASH model can be manipulated programmatically via actions. An action represents a transaction in which a series of atomic actions are performed. Atomic actions are operations on a live ASH model, and include the following:

- createNode(type, parentId, offset) - creates a new node instance and adds it to the stack, returns the node's id (it's index in the stack).
- uncreate node - pops the top Node off the stack and destroys it (frees all resources associated with it).
- setProperty(id,property,value) - sets the given property value.
- unsetProperty(id,property) - unsets the given property of the given node (resets it to its default value when no value is specified).

ASH plugins (node type implementations) must implement the following functions:

- xmlns() The XML namespace to which this node type name belongs.
- typeName() The name of this node type
- addChild(Node child) Adds a child node to this node.
- removeChild()

An ASH session state can be serialized as XML.

ASH plugin repositories map ASH node type names to URLs from which the implementing JAR file can be downloaded and a list of plugins it depends on.