Data Ontology Cache System
The Monadics

Derek Mansen    Marc Weil    Matt Kotsenas
Robbie Gladmon  Tom Rudick

Two Sigma Investments LLC.
Mark Roth

Faculty Mentor
Dr. J. Scott Hawker

Project Overview

The goal of the Data Ontology Cache senior project is to create a data ontology system to discover and exploit relationships between data gathered from disparate sources, and to enable Two Sigma to develop better business insights from existing data and systems. Two Sigma is responsible for retaining large volumes of data (in the petabytes), and it is very difficult to obtain a single, cohesive view of its operations. Questions such as "are we purchasing financial datasets that no analyst is using?" are very difficult to answer. This project should solve this problem and allow Two Sigma to use the information at its disposal more easily and efficiently.

The system consists of several components: the Collector API, Ontology Core, Query and Reporting Engine, Command Line Tools, and Web Interface.

The Collector API is responsible for aggregating all data sources into a common data format. This data is then stored in the ontology cache. The Query and Reporting Engine is used to query and create reports based on the knowledge base stored within the Ontology Core. The Command Line and Web Interfaces interact with the Query and Reporting Engine and provide a convenient method for interacting with the system. Finally, a security manager is responsible for enforcing both document and user-level security concerns. Semantic web technologies such as OWL [1] and RDF [2] are used as the building blocks for many of these components.

The final product is a system that allows users to discover previously unknown relationships between artifacts and develop previously unavailable business insights.
Basic Requirements

The Collector API is a flexible entry point for data collectors to push ontology data into the Ontology Core. A data collector is defined as anything that gathers data from any number of different sources and converts that data to a common format that the Ontology Core can accept. The Collector API provides a convenient way for data collectors to push RDF/XML into the Ontology Core. For security purposes, collectors must identify themselves and provide authentication information in order to push data into the core.

The Query and Reporting Engine provides an interface for the users to extract information from the Ontology Core once it has been pushed by a data collector. The system also provides an interface to generate reports. These reports can gather data from the Ontology Core and format them as necessary.

The Web Interface of the project consists of several ways to visualize data within the system, along with a suite of basic administrative tools. Users should be able to perform full-text searches over the data with a minimalistic interface, and view results in a similar manner. Reports should be runnable through the web interface, and configurable by an administrator. The web interface must also provide a tree-like, hierarchical structure, which groups query results into folders. Administrators can configure the queries provided by the tree interface.

The system provides Command Line Tools for common tasks, such as collection, querying, and administration tasks. These tools are provided so that tasks can be scheduled to run automatically using shell scripts. These tools behave much like common Unix tools, and should follow the same paradigm (one tool per task, pipe-able output, etc.).

The Ontology Core is the main back-end component of the system. It contains a cache of data, pushed by data collectors. The ontology core accepts collector data pushes and queries, and returns collection reports and query results for each, respectively. The core keeps track of which collectors have pushed data into the core. When a collector pushes its next set of data, the Ontology Core replaces all data that had been previously pushed by the same collector.

The data cache stored by the Ontology Core needs to be secure. Each node in the data graph can have an optional security group property added to it. When a user queries and a secured node is part of the result set, the core must check to see if a user belongs to one of the groups specified by this security group property. All results are filtered through this layer of security. By default, all nodes are accessible by all users unless the security property is explicitly set. Collections are secured using authentication data which is passed to the Ontology Core at the time of collection.

Performance of the Ontology Core is of the utmost importance to Two Sigma. A common use case of the system involves an analyst browsing the data in the core, discovering new
relationships by executing different queries, or exploring visualized relationships. To this end, querying the ontology core should never take more than 1-2 seconds.

**Constraints**

When the project first began, Two Sigma provided a number of constraints on the technologies used to deliver the product. Below is a list of the original constraints:

- Any open source libraries used must have a commercial-friendly license (acceptable licenses include BSD-style licenses and specifically exclude GPL, though LGPL may be acceptable in some circumstances)
- Ontology Core and web environment must run on Ubuntu Linux
- Must be implemented in Java 6
- Must provide OWL or RDF export capability for data
- Relational databases must be deployable using Microsoft SQL Server 2008
- The Web application must be implemented using Google Web Toolkit
- The Web application must be deployable to a Tomcat web container
- Preferred scripting language for views and reports is Groovy using SPARQL to do semantic queries
- The Web application must be compatible with both Internet Explorer 7 and Firefox 3

As a result of discussion with Two Sigma, the Google Web Toolkit constraint was removed.

Other constraints were provided as guidance from Two Sigma. The initial architecture was developed from a diagram that was provided in the project overview. We were given recommendations for technologies to use, such as Jena [3] for the ontology core. Each technology was evaluated, and it was up to us to decide if the technology was a good fit for the product. We always asked Two Sigma for approval on these technologies, and they were always very open to our decision process.

**Development Process**

As this project was very research-intensive in terms of the initial learning curve of many of the technologies used, a two-stage development process was used to mitigate the risk of not knowing the technology domain. Two Sigma was open to the team creating the development process for the project; there were no mandates for process other than regular status updates and reviews of artifacts. The first half of development (ten weeks) was designated as the "research" phase of the project, and the second half was designated as the "implementation" phase. Using unfamiliar Semantic Web technologies carried large technological risks. The goal of the research phase was to drive down these risks and to become familiar with the technologies. During this time, proof-of-concept demos were created for each proposed technology. The product at the end of this first half was a fully-functioning slice of the desired features; each subsystem was represented and shown to be feasible for the project's requirements.
Once technology risk was driven down to a satisfactory level, the process changed to an iterative release schedule. The goal of these iterations was to increase the value of the product by adding features that satisfied the requirements. Given that the time frame for this portion of the project was only ten weeks, iterations were scheduled in two-week blocks with a product release at the end of each iteration. This process was chosen because the requirements for the project evolved through prototyping and communication with the sponsor. At the end of each iteration, Two Sigma would be given a new release and requirements validation would be performed. At the outset of each iteration, activities were estimated and assigned to team members. Features were prioritized by estimated risk and value. Each team member was then responsible for completing the assigned tasks by the end of the iteration.

The Trac [4] project management tool was used to help monitor the status of the project through its life cycle. Most of the Trac features used were designed to aid in high-speed development. For the research phase, the Trac wiki was used heavily, as it allowed easy concurrent and collaborative editing. Two Sigma was also given access to Trac so they could monitor progress and aid with gathering information. Once the team entered the implementation phase, all tasks were recorded as tickets in the system, along with the assignee, the estimated hours, and the iteration to which it belonged. This ticketing system allowed easy tracking of hours, along with time spent completing features for each iteration.

On the development side, the Mercurial Version Control System [5] was used to track changes. We also used the Hudson [6] build server to run automated builds of the project whenever new changes were pushed to Mercurial. The Ant [7] build tool automatically
ran all unit tests and generated code coverage reports based on those tests. Using this system helped us keep a working set of the code in source control at all times.

Communication with Two Sigma was designed to be as transparent and lightweight as possible. Weekly video conferences addressed requirements, problems the team was having, and showcased the progress of the project through demonstrations and status updates. Each meeting was focused around an agenda that was shared in advance to minimize idle chatter.

**Project Schedule: Planned and Actual**

Because of the brevity of the senior project experience, time was an important consideration when planning the project schedule. Additionally, the scope, domain, and technologies involved with the project created a very high risk of schedule overrun. To compensate for this risk, the team incorporated some ideas from time-boxed development to keep the project on schedule and headed towards success. From Steve McConnell's "Rapid Development" [8], we decided that limiting features to iterations would help prevent schedule slip. Time-boxed development prevents schedule slip by requiring re-evaluation and re-prioritization of the remaining work at the end of each iteration. From the combined set of planned activities, the team would then discuss and consult with the project sponsor to decide what activities must be removed from or added to the remaining project schedule to fit the new estimates. This process for controlling the schedule relies heavily on the accuracy of estimates to keep activities within their respective iterations and to re-evaluate the new scheduled based on the new estimates.

Though different processes were used in each half of the project, using time-limited iterations aided the team in both instances. During the first half of the project, the earlier iterations were focused on understanding the technology domain and evaluating tools in order of importance so design decisions could be made. The initial iterations focused on understanding the requirements put forth by the project sponsor. Our research led us to produce proof-of-concept prototypes by the end of the first half of development.

The second half of the project focused on driving up the value of the product delivered by implementing features. Throughout this half of the project, features were prioritized and developed in specific iterations. If any features could not be completed in their designated iterations, the schedule was adjusted to accommodate schedule slip. Each iteration then became a milestone and a stable product was released to Two Sigma for internal testing. This release schedule allowed for more frequent customer feedback during development. Feedback, in terms of change requests and bugs, was estimated and worked into the schedule for a future iteration. End-of-iteration meetings allowed time to review estimates, remaining project activities, and make appropriate revisions to the schedule. These revisions were necessary because some activities took less time than estimated; this extra time was used on items from the next iteration. Pulling activities from later iterations, adding bugs, and updating items with changes led to a dynamic schedule that changed throughout the second half of the project. In total, this scheduling method was effective for the project and instrumental in achieving success.
System Design

System Architecture

Our decision to distribute the system across multiple nodes and communicate between these nodes via Apache ActiveMQ [9] was an early design choice that had far reaching implications in the design of the system. We also chose to route most of our inter-thread communication through this message broker. This decision was made because we believe that large systems that communicate through explicit locking and thread synchronization quickly become difficult to maintain, and we wanted to avoid that complexity. This decision simplified communication between threads and distributed nodes.

The diagram below shows the high-level architecture of the system. The dotted lines indicate deployed node borders. Communication between these systems happens through ActiveMQ.

Ontology Core

Technologies and Formats

The standard format for data entering the system is RDF/XML, as this format is generally used for semantic data on the Internet. The library that chose for storing and manipulating ontology data is called Jena. Jena provides methods for reading RDF/XML data and making queries on that data. We use another library called TDB [10] for persisting the data in the cache. Internally, the data is represented as a set of named graphs [11]. Each graph stores the data pushed by a single collector. When the same collector pushes a new set of data, the old graph is dropped from the set and the new graph replaces it.

The standard language for querying RDF data is called SPARQL (SPARQL Protocol and RDF Query Language) [12]. All queries must be constructed using SPARQL.
Since there are existing libraries in the Ontology Core that could limit performance, the Ontology Core can be replicated. ActiveMQ can load-balance queries across all cores. Performance of collections is not as important as that of queries.

Overview

The Ontology Core has several layers, each of which performs a specific function as a query or collection is passed through the system. The Listener layer has the responsibility of interacting with ActiveMQ and receiving messages. No other component of the core should communicate directly with the messaging system. The classes contained in this layer handle receiving collections, receiving queries, and coordinating replication data.

The Dispatcher layer is the interface that organizes requests for access to the underlying Jena ontology model. The job of any class that implements the Dispatcher interface is to ensure that there are no concurrency problems with the model, as Jena is not safe for multi-threaded access under all conditions. Currently, the system is single threaded. If Jena's access model changes in a future release, a new Dispatcher could be created that handles access in a different way.

The SecureModelAccessor holds the reference to the underlying ontology model. This class is the only way to interact with the underlying Jena ontology model. It enforces ontology-level security for queries, and processes collections as they enter the system. It is important that the Jena model is secure, so this layer handles all access to that model.
Listeners

The system of listeners that receive and respond to messages from ActiveMQ comprises most of the logic in the ontology core. Each listener follows a pattern that is defined in the Listener abstract class as a template method. The pattern has three steps: receive a message from ActiveMQ, process that message, and send the response message to ActiveMQ. Each listener always receives from the same ActiveMQ destination, but it can set a response destination depending on how the message was processed. The interaction template is shown in the sequence diagram below.

By routing all inter-thread communication through ActiveMQ, complexity in code is traded for complexity in the ordering and processing of messages. We have chosen to define these in more detailed sequence diagrams that will not be reproduced here. Using ActiveMQ made the increased complexity of core replication manageable.

Collector API

The Collector API exposes functionality related to data collectors and related tasks. Our goal when designing this API was to make it as simple as possible for developers. Per the requirements, when pushing collection data, the collectors generate a string or stream of RDF/XML data. The methods for pushing data are simple, taking only the data (in either String or InputStream format), the collector name, and any metadata associated with that collector. In addition to pushing data, the API also provides methods for sending a new schema to the Ontology Core.

The API is provided as a small JAR file that contains the CollectorControl class (which has all relevant methods) and necessary ActiveMQ libraries. Two Sigma can then write collectors which link to this library.
Query API

The Query API exposes functionality related to querying the ontology cache. The query method takes a SPARQL query and a SecurityToken, and returns the set of results of that query. The core uses this token to find out which security groups the user belongs to, if any. The security classes are not fully implemented, as we did not have access to Two Sigma's internal security system. Two Sigma can integrate its existing single-sign-on system within these classes. As with the Collector API, the goal was to provide a clean, simple interface. The delivery for the Query API is another compact JAR to which query clients can link.

Web Component

The Web component is based on the Spring MVC [13] framework, and uses the Query API along with Data Access Objects to retrieve data from the Ontology Core. Several Spring-annotated Java classes act as the controllers. These controllers accept and process requests for various web resources. When a request is dispatched to the web server, Spring-MVC runs the appropriate controller code. This code then accesses a predetermined model object and responds to the request.

Each web view is a rendered Java Server Page. The JavaServer Pages Standard Tag Library [14] is used to perform basic processing on data within a Java Server Page. JavaScript is also integrated into the pages in order to perform AJAX queries and make some components of the web project more responsive.

There are three main controllers in the web project: Search, Reports, and Trees. Each one of these controllers contains the logic to perform all the operations in a specific web section. Additionally, there are a number of controllers that correspond to various administrator pages. These controllers are much smaller than the three previously mentioned and usually only run a single operation.

Command Line Tools

Many of Two Sigma's internal processes use chains of Unix-style command line tools. The bundled command line tools replicate the relevant portions of the administrative functionality of the web interface, such as exporting a full dump of the data cache, running Groovy reports, and viewing the status of all the running cores. To avoid coupling the command line tools to ActiveMQ and duplicating the related communication code already present in the web application, the command line tools simply send HTTP GET and POST requests to a running web application instance. These tools make use of the RESTful URI scheme and full range of HTTP verbs used by the web application for creating and retrieving resources in different human- and computer-readable formats such as CSV, XML, and JSON.
**Groovy Reports**

Internally, Two Sigma uses the Groovy [15] language, a dynamically-typed language built on top of the JVM [16], along with its built-in templating language to create nicely formatted reports from their different sources of data. Our system includes a Groovy binding layer that allows users to reuse their knowledge and easily create reports from data stored in the data cache.

Reports can be run in two different ways: using a provided command line tool or using the web application. The web application allows administrators to upload Groovy scripts and grant execution access to other users. In either case, users can also configure variables; when the report is run, the values of those variables are injected into the script so they are bound when referenced. The command line tool is intended for testing and private reporting purposes since reports executed using it are not saved.

MIME types [17] can also be specified in either tool. These types allow reports to return data formats other than text or HTML. For example, a report author may want to generate an image using a mapping API and a set of mailing addresses stored in nodes in the cache. Using Groovy's templating capabilities, reports can be output in any format such as HTML, CSV, or PDF.

When writing reports, an instance of the Query API is bound to a variable with a pre-specified name. Report authors then use this object to send raw SPARQL queries to the Ontology Core, which responds with an instance of the result set class, which is also defined in the Query API.

**Metrics**

**First Quarter Metrics**

- Requirements volatility
- Risk volatility
- Hours worked

As stated earlier, the first quarter of the project revolved around research and feasibility studies, not delivering value. As a result, process metrics were favored over product metrics to ensure that the team remained on schedule and avoided the dreaded "analysis paralysis." Our risk volatility, unsurprisingly, started very high and was driven down to an acceptable level. Requirements volatility was lower in the first quarter than it would most likely be in other situations because the main goal of the first quarter was to understand the domain, not to solidify requirements.
Second Quarter Metrics

- Project velocity (Planned feature points completed per iteration)
- Estimated vs actual hours per feature
- Hours worked

The goal of the second quarter was to deliver value to Two Sigma, so our metrics focused on maximizing value given the time allotted. In order to support this goal, we chose metrics and measurements that focused on schedule and task estimation. Recall that each iteration was measured in terms of both hours worked and project velocity, which was calculated as feature points per iteration. After each iteration, project velocity was recalculated, incorporating the newly available schedule evidence, and the remaining schedule was adjusted to assume this newly calculated velocity. As a result, the team had a consistent and manageable view of the remaining work in both the current iteration and the project as a whole.

Product State at Time of Delivery

All major features were delivered in the final product to Two Sigma. As with any software project, the vision of the features changed; some increased in complexity and scope while others decreased. Feature change was driven by the current schedule, the value of the feature, and value of other features yet to be implemented. At the end of the development iterations, we demonstrated delivered features to Two Sigma during weekly video teleconferences and obtained feedback and refined feature definitions. In addition, smaller features were added to the project at the team's discretion. These additions include convenience and informational features, such as adding an indicator on the Core Status web page to denote the type of cores running.

The final product was delivered in a stable state. All known issues or limitations relating to tool choices and design decisions have been documented. This documentation also includes information on what interfaces needed to be implemented in order to integrate with Two Sigma's existing security systems. These implementations exist in both the web interface and the core. Though the delivered product does enforce security, it does not perform authentication of users, per the agreement with Two Sigma, as little value would have been added by integrating security compared to the amount of implementation time required.
**Project Reflection**

We succeeded in delivering a product that met Two Sigma's requirements. As a team, we quickly became proficient in the numerous technologies that were used in the project. We made each process decision with the tight schedule and high technical risks that we faced in mind. Our process took some time to develop, but by the second half of development, it became second nature to us. The time-boxed iteration process became indispensable by the end of the project, and kept us on schedule.

Our tests were not as robust as we would have liked because of the distributed nature of the system, as well as the fact that much of the core code depended on external libraries. We were also not able to develop realistic data for performance testing, due to unknown characteristics of Two Sigma's proprietary data. We relied too heavily on Two Sigma for sample data, which they were unable to provide.

We did not spend as much time as we would have liked gathering and analyzing measurements for our product metrics. Project velocity and hours worked per iteration were very useful metrics in terms of scheduling. However, product metrics such as cyclomatic code complexity or defects per KLOC were of lesser value due to the "glue code" nature of much of the project.
References