12-2014

A Deep Search Architecture for Capturing Product Ontologies

Tejeshwar Sangameswaran
University of Arkansas, Fayetteville

A DEEP SEARCH ARCHITECTURE FOR CAPTURING PRODUCT ONTOLOGIES

A DEEP SEARCH ARCHITECTURE FOR CAPTURING PRODUCT ONTOLOGIES

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Tejeshwar Sangameswaran
University of Arkansas
Bachelor of Science in Computer Engineering, 2010

December 2014
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____
Dr. Craig Thompson
Thesis Director

_____
Dr. Gordon Beavers
Committee Member

_____
Dr. Bajendra Panda
Committee Member

ABSTRACT

This thesis describes a method to populate very large product ontologies quickly. We discuss a deep search architecture to text-mine online e-commerce market places and build a taxonomy of products and their corresponding descriptions and parent categories. The goal is to automatically construct an open database of products, which are aggregated from different online retailers. The database contains extensive metadata on each object, which can be queried and analyzed. Such a public database currently does not exist; instead the information currently resides siloed within various organizations. In this thesis, we describe the tools, data structures and software architectures that allowed aggregating, structuring, storing and searching through several gigabytes of product ontologies and their associated metadata. We also describe solutions to some computational puzzles in trying to mine data on large scale. We implemented the product capture architecture and, using this implementation, we built product ontologies corresponding to two major retailers: Wal-Mart and Target. The ontology data is analyzed to explore structural complexity and similarities and differences between the retailers. A broad product ontology has several uses, from comparison shopping applications that already exist to situation aware computing of tomorrow where computers are aware of the objects in their surroundings and these objects interact together to help humans in everyday tasks.

ACKNOWLEDGEMENTS

I express my sincere gratitude to my thesis advisor, Dr. Craig Thompson. I thank him for his patience, guidance and support from the start to the end of my research. Without him, this thesis would not have been possible. I also thank my thesis committee members, Dr. Gordon Beavers and Dr. Bajendra Panda for their time and advice in reviewing this thesis document.

I also thank my parents and my wife for their love and support throughout my degree program.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Context

Imagine walking up to a thermostat; you pull out your cell phone; and it automatically pairs with the thermostat. Communication interfaces are exchanged and you can instantaneously control the thermostat via your smartphone. Imagine doing the same with every smart networked object in the world. In that smart semantic world of the future, this will be the norm.

With the advent of Internet of Things (IoT) and smart objects, the word around is becoming increasingly computationalized. Unfortunately, there is yet to be a standard set of communication protocols for smart objects. So, almost all smart objects have proprietary interfaces, and, at present, there is no common way for these devices to interoperate and communicate freely with each other. So, at present, humans have to learn several different query and command interfaces to talk to the few smart objects that so far exist.

In the predicted future smart semantic world, all of the objects around us will be networked and will use common protocols to communicate freely with each other [1]. There will be a universal "soft controller" (e.g., near future generation smart phones or smart glasses) that will provide a common mobile interface to these objects. Common interfaces will expose a standardized API that will facilitate easy interoperability and extensibility. This would help create a world where all objects and humans can work in cooperation with each other. Building such a universal controller is no small task, but this thesis takes a few steps towards that grand goal.

1.2 Problem

In the future smart world, for two objects to interoperate with each other, they must first share a common ontology, that is, a common way to represent attributes, interfaces, and a common category type system [2]. To build a common ontology requires a categorization and classification of *all* objects around us. The process of creating of this organizational knowledge will give us insights into building abstraction and inheritance interfaces for these objects. Unfortunately, an open database containing a comprehensive hierarchical organization of items in our everyday world does not exist. This thesis is a half-step toward such an architecture that can automatically build such a common product ontology. We will extract and analyze ontology data sets but we do not yet merge them into a "super ontology."

1.3 Thesis Statement

The primary objective of this thesis is to describe a software architecture that can structure a large amount of existing real-world data. The scope of the ontology we will describe is everyday retail products, the universe of man-made things (not including at this time objects in nature or abstract objects). This structured data will be represented in a very large, open graph of everyday objects, arranged in hierarchy of their ontological relationships. Each object in this collection will be augmented with relevant descriptive metadata.

Making an exhaustive list of products around us seems like a daunting task, but in fact most of the man-made items around us can be purchased online through the websites of various retailers. If we can aggregate products from all the retail stores, that will provide us a good first approximation of all products around us.

This thesis addresses the following issues:

- Interfacing with various retailers: Extraction and classification of their product catalog ontologies from various retailers
- Storing and indexing of semantic data: For ad-hoc querying of this large knowledge base.
- Comparative analysis of ontological hierarchies of different retailers.

Design criteria for the architecture include coverage and efficiency and will be used to judge the architecture in the Conclusion section 5. Specifically, the criteria are:

- Semantic Coverage – the ability of our representation formalism to adequately represent all of the metadata about entities, attributes, and relationships in the retailer website.
- Coverage Completeness – the ability of the ontology database to adequately represent all the instance metadata from the retailer website.
- Traversal Efficiency – the efficiency of traversing the retailer website to build our product ontology.
- Query Access Efficiency – the efficiency of queries to access product data from our ontology.

1.4 Approach

Retailers keep their product catalog online and available to humans but often do not provide a way for computers to access or search this information. As such, this data can be viewed as part of the "dark web" that is not directly available to search engines [3]. It is unlikely that computers would be able to get direct access to company inventories and ontologies. Some retailers do provide an API. Wal-Mart for example has a product API [4] that allows programs to

retrieve metadata about a product given its unique Wal-Mart specific ID. But to use their API, a developer has to create an account on the Wal-Mart development platform and get a unique developer key. This key has to be appended to every API request for a resource. Wal-Mart effectively then cripples this service by enforcing a hard limit of fifteen queries per hour for a given developer key. These limits are so restrictive that to download metadata about every item on the Wal-Mart catalog of roughly 2M items would take 11.5 years. Clearly this is not feasible.

The only guaranteed way to get a live feed from an online retail catalog from a major retailer is through their customer facing e-commerce portal using screen scraping. It is hard for retailers to impose restrictions on automated bots crawling their site. We built a guided web crawler that can dig through the dynamically generated web pages of a retailer and build a taxonomy of products from the information. This process had to be parallelized because of the sheer amount of data that is to be processed.

We then created an in-memory inverted index of the products scrapped from these sites, so that we could perform ad-hoc search queries. We represented the ontological data as a directed graph structure, and applied graph algorithms to gain insights into the complexity of the organizational structure.

1.5 Organization of this Thesis

Chapter 2 of this thesis deals with some background information on the topics discussed in this thesis. We discuss existing work and how this thesis relates.

Chapter 3 deals with the high level architecture of the retail web crawler and some specific implementation details including providing details about each service, their inputs, outputs, and service contracts. We also discuss some of the technical challenges and solutions

Chapter 4 analyzes aggregated retail data compiled by the software system of Chapter 3.

Chapter 5 provides a summary and identifies future work that can be done to further the project.

2. BACKGROUND

2.1 Key Concepts

This thesis is builds on interdisciplinary topics including knowledge representation, ontology, Internet of things, smart semantic world, virtual worlds and mirror worlds, and deep web. This section provides some background information on these topics.

2.1.1 Knowledge Representation

Knowledge representation is a field as old as Greek philosophers Plato and Aristotle and is central to artificial intelligence. For Plato, a Platonic Form is an idealized concept of a real world thing – the *chairness* that defines all real world chairs. Aristotle's categories were an attempt to represent the world around him into discrete categories. He believed all objects across all domains could be generalized into categories. His categories are of predicates, which describe the world [5].

Humans intuitively understand the nature of things. We can keep track of objects and their interfaces. For example: When we are referring to a piece of furniture in our living room, we say a chair or perhaps even a recliner; but we don't say SKU# 4412-321, even though we can distinguish between different kinds of recliners. Retailers similarly distinguish between kinds of chairs even if they do not use category names for each subtype. Instead, they use Stock Keeping Units (SKUs) and can identify different descriptions (tech specs) with each as well as different prices. Where humans intuitively reason that the recliner has a super class of chair, computers do not have this ability but instead they require explicit categories. Knowledge representation

deals with the formalization of the vast amount of default knowledge that we as humans take for granted.

Knowledge representation is in a way the bridge between artificial intelligence and ontologies, which allow computers to reason about the world around them [6].

2.1.2 Ontologies

Tractatus Logico-Philosophicus is a book by the famed 20th century linguistic philosopher Ludwig Wittgenstein which is considered to be a seminal work in the field of ontology and the nature of representation. In it, he defined the world ontology as a "formal, explicit specification of a shared conceptualization" [7]. Ontology deals with trying to understand the nature of things and the challenge of categorizing and representing them. Ontologies try to create a hierarchical representation of what exists in the world around us. There is no concept of "one true ontology", instead hierarchy is usually domain specific and based on specific traits and the attributes of the object.

Why go through the trouble of cataloging and categorizing all the objects around us? What are the benefits the ontology? Pichler and Weber provide a well-reasoned answer to his question in their paper "Sharing and Debating Wittgenstein by using an Ontology" [7]

> (1) Limitations of free text search are overcome, as semantic labels allow for searching and browsing by concept rather than string-based only; (2) a grouping of these concepts, which relates them to each other and organizes them into classes and subclasses; and (3) drawing inferences and reasoning become possible, as both the human and the machine will be able to extract from the specific position of a concept information about its place in the overall conceptualization.

Having a formal representation of the relationships between objects will help us in building a computational model of the world around us. This computational model can be queried and operated upon. We can apply object-oriented principles on these ontologies and design abstract classes and inheritance patterns for the objects around us.

2.1.3 Internet of Things

Internet of Things (IoT) primarily deals with trying to get devices in our homes and offices to interface to an interconnected network so that they can be operated and controlled remotely [8]. The majority of the value will arise not from just a single item being connected to the Internet, but the networked devices working in unison.

These objects that are Internet-connected are referred to as "Smart Objects". They often have some form of sensor attached to a microcontroller and can log an extensive amount of data and provide analytics and insights into our habits and our lifestyles.

Figure 1: Some "Internet of Things" Smart Objects currently on market

2.1.4 Smart Semantic World

Smart semantic world is a term coined in the 2011 Eguchi and Thompson paper titled "Towards A Semantic World: Smart Objects In A Virtual World" [1]. The term merges the terms *smart world* coined by IBM and *semantic web* coined by Tim Berners Lee. The Internet of Things revolution will eventually lead to a Smart Semantic World where objects are aware of themselves and other objects around them [1]. All smart objects will have well defined interfaces and follow a standardized communication protocol that facilitates interoperability. In this future world, objects will freely communicate with each other to make intelligent decisions to improve the quality of life.

An example, while at work, you make a status update on Twitter about how you think you are about to catch a cold. This information is processed and communicated to the objects around in your home. Your smart thermostat automatically warms up the house. Your smart light bulbs re-program to provide soothing dim lights. Your drink maker makes you cup of hot tea. Your car informs your house when you will be home so all of this waiting for you upon arrival.

2.1.5 Virtual Worlds and Mirror Worlds

Virtual worlds are immersive 3D graphical environments in which avatars controlled by humans can interact with the environment and each other. The most popular of these is Second Life developed at Linden Labs.

Second Life provides a completely editable and programmable environment. Users can assemble building blocks graphical primitives called "prims" into more complex objects. Interaction logic can be programmed directly into each object.

Figure 2: A Hospital in Second Life[1]

Mirror Worlds are when the real world objects, environments and workflows are modeled inside a virtual world to create a functioning replica of the real world. This allows complex simulations to be run to build models. These models can then be used to make decisions in the real world. Mirror worlds inside of virtual worlds are useful in the fields of logistics, supply chain management, and healthcare.

Eno and Thompson's 2011 paper titled "Virtual and Real-World Ontology Services" describes how virtual worlds can be used to build detailed ontologies of the real world. Leveraging virtual world data structures can create a model of the real world. Mirror worlds objects are highly detailed; they contain images, 3D models and labels. These can be translated to the real world and be used to define objects in the smart semantic world [9].

2.1.6 Deep Web

Search engines like Google send their web crawlers (aka harvesters, spiders) out to harvest and index static web pages. But it has long been recognized that there is a dark or deep

[1] Courtesy http://vw.ddns.uark.edu/index.php?page=overview

web that contains considerable information that is not accessible to traditional search engines. In some cases, organizations publish application program interfaces (APIs) to data reservoirs enabling computers to access this kind of data. Another approach is to build screen scrapers that dynamically visit pages that are themselves constructed from database data available and dynamically constructed upon request.

An analogy can be drawn between virtual worlds and the deep web. Second Life has a vast amount of information that is not indexed. This information includes ontological classifications and 3D models of objects in the virtual world. Once this knowledge is captured, it can be translated to the real world. However, this data cannot be crawled through conventional web crawlers. Specialized crawlers which are capable of exploring and traversing the 3D world need to be built [9]. These crawlers query each object they visit and build a queryable model of the world.

2.2 Related Work

2.2.1 DARPA Agent Markup Language (DAML)

DARPA Agent Markup Language (DAML) was a US Department of Defense research program (1999-2002) to pave the road towards the semantic web. It aimed to use XML or RDF to provide “semantic grounding” for the web [10].

```
<Property ID="father">
  <subProperty resource="#Parent"/>
  <range resource="#Man"/>
  <cardinality>1</cardinality>
</Property>


<UniqueProperty ID="mother">
  <subProperty resource="#Parent"/>
  <range resource="#Woman"/>
</UniqueProperty>

<Property ID="mom">
  <equivalentTo resource="#Mother"/>
</Property>
```

Figure 3: Sample DAML document

This model contained agents, which performed units of work. The agents use a form of annotated XML called DAML to communicate amongst each other and to make queries. DAML (and later DAML+OWL) had special tags to denote relationships and other metadata. These agents were made aware of each other and had excellent interoperability because they operated under a common ontology [10].

2.2.2 Everything is Alive

Everything is alive (EIA) is an ongoing research program headed by Dr. Craig Thompson at the University of Arkansas. The project focuses on creating a set of networked world of smart objects that sense, act, think, feel and communicate [11]. Real world smart objects have sensors attached to a microcontroller. The project used 3D virtual worlds, e.g., Second Life, to model real world places and situations. Second Life uses Linden Scripting Language (LSL) to enable

objects to have logic attached to them. This allows mimicking of real world behavior of smart objects in these virtual worlds.

One goal of this project is to demonstrate how to create a 1:1 mirror world replica of portions of the real world inside Second Life. Objects in the real world are tagged with an RFID tag which is used to correlate data with its virtual counterpart inside of Second Life.

One of the projects under the EIA umbrella created a detailed virtual hospital inside of Second Life (dubbed a "Hogspital" in honor of the University of Arkansas razorback mascot). Simulations run inside this hospital have been used to study "cost reduction, improved safety, better visibility, better performance, and more automation." [12] in real world hospitals.

2.2.3 DBPedia

The DBPedia project is striving to make Tim Berners Lee's dream of a semantic web into a reality using "linked data" which is data that can represent relationships among web objects. It is a community-driven effort to effort to build a clean and structured database of the vast amount of ontology information currently on Wikipedia. The projects goal is to restructure Wikipedia information (which is designed to be consumed by human) into a format that is machine-consumable [13]. This will allow the data to be queried by intelligent systems. Artificial intelligent system will be able to make connections, which were previously unknown.

The project has made considerable progress in the last few years, amassing gigabytes of structured data.

> 4.0 million things, out of which 3.22 million are classified in a consistent ontology, including 832,000 persons, 639,000 places (including 427,000 populated places), 372,000 creative works (including 116,000 music albums, 78,000 films and 18,500 video games), 209,000 organizations (including 49,000

> companies and 45,000 educational institutions), 226,000 species and 5,600 diseases. [13]

2.2.4 Amazon Firefly

Amazon released their own Smartphone dubbed ‘Fire’ in June 2014. What sets this phone apart from the already saturated Android market is the FireFly feature:

> Firefly works like this: you press the Firefly button on the phone’s left side, and when the app pops up, you can immediately start identifying things like printed text in posters, get information on movies and TV episodes, recognize music and over 70 million products in Amazon’s stores, including household items, books, DVDs and more. It pretty much turns your device into a scanner that can quickly direct you right to the product you identified on Amazon’s website. [14]

This project allows a person to a dedicated button that activates image understanding to lookup any item in their immediate surrounding on Amazon’s website. Amazon’s website has extensive details on the product; they price, description, product specific meta data and related products. We can see how the product is classified in Amazon’s detailed product ontology.

Figure 4: Amazon Fire running the FireFly app[2]

2.2.5 Existing Ontology Work

A few attempts in building product ontology have been made. Although none of work was done to advance towards the goal of a smart semantic future, they share a common methodology. In the paper titled "Practical Issues for Building a Product Ontology System", Ig-hoon Lee et al. explore a way to build product a product ontology by aggregating data from e-procurement services, websites of distributors, and way for Customers to directly input the information [15]. The PCS2OWL project aims to build product classification systems as web ontologies [16]. They do not focus on actually building the product ontology, but instead focus on standardizing product ontologies by converting different ontologies into a standard language such as the OWL semantic markup language.

---

[2] Courtesy http://www.amazon.com/Fire_Phone_13MP-Camera_32GB/dp/B00EOE0WKQ

3. ARCHITECTURE AND IMPLEMENTATION

3.1 High Level Design

A screen-scraping approach is used to collect the retail ontology information that is currently publically available but stored in the deep web as dynamic HTML pages. This section describes the architecture and implementation of the system to access and build the product ontologies.

The architecture of this software is based on the service-oriented architecture (SOA) guidelines: "*separation of tasks"*, "*loosely coupled services"* and "*well-defined interfaces"* [17]. The software is decomposed into several small services, each of which perform a single operation and have a strict service contract. This pattern forces separation of concerns and helps keep the size of codebase small. Scaling is straightforward with this type of architecture - simply instantiate more instances of the service.

To fully reap the benefits of SOA, the principles dictate that services do not talk directly to each other, but rather communicate through a centralized service bus. This centralized bus is a robust piece of software that load balances messages between all the services. It enforces queue order, provides a durable message brokering service and provides several middleware integration patterns out of-the-box [17]. Most SOA busses are developed my multi-million dollar corporations with profitability in mind. So, these busses are expensive, often costing hundreds of thousands of dollars in licensing costs [18]. There are actively developed open source alternatives: Apache Service Mix [19] and Apache Synapse [20]. But they have proven to be too cumbersome and add too much complexity to the infrastructure. Deploying these services would

require a fleet of servers for redundancy and a technically skilled operations team to maintain them. So, the architecture was retooled to operate without a centralized bus.

Instead of having a bus that actively routes messages between each service, there is a shared storage area between the services. Each service writes its output to a data store. The next service consumes from this data source, processes the information and writes its output to another data store which feeds another service and so on. This waterfall method of sharing data allows for high modularity and therefore scalability. It allows each service to function separately from the others. This architecture is illustrated in Figure 5.

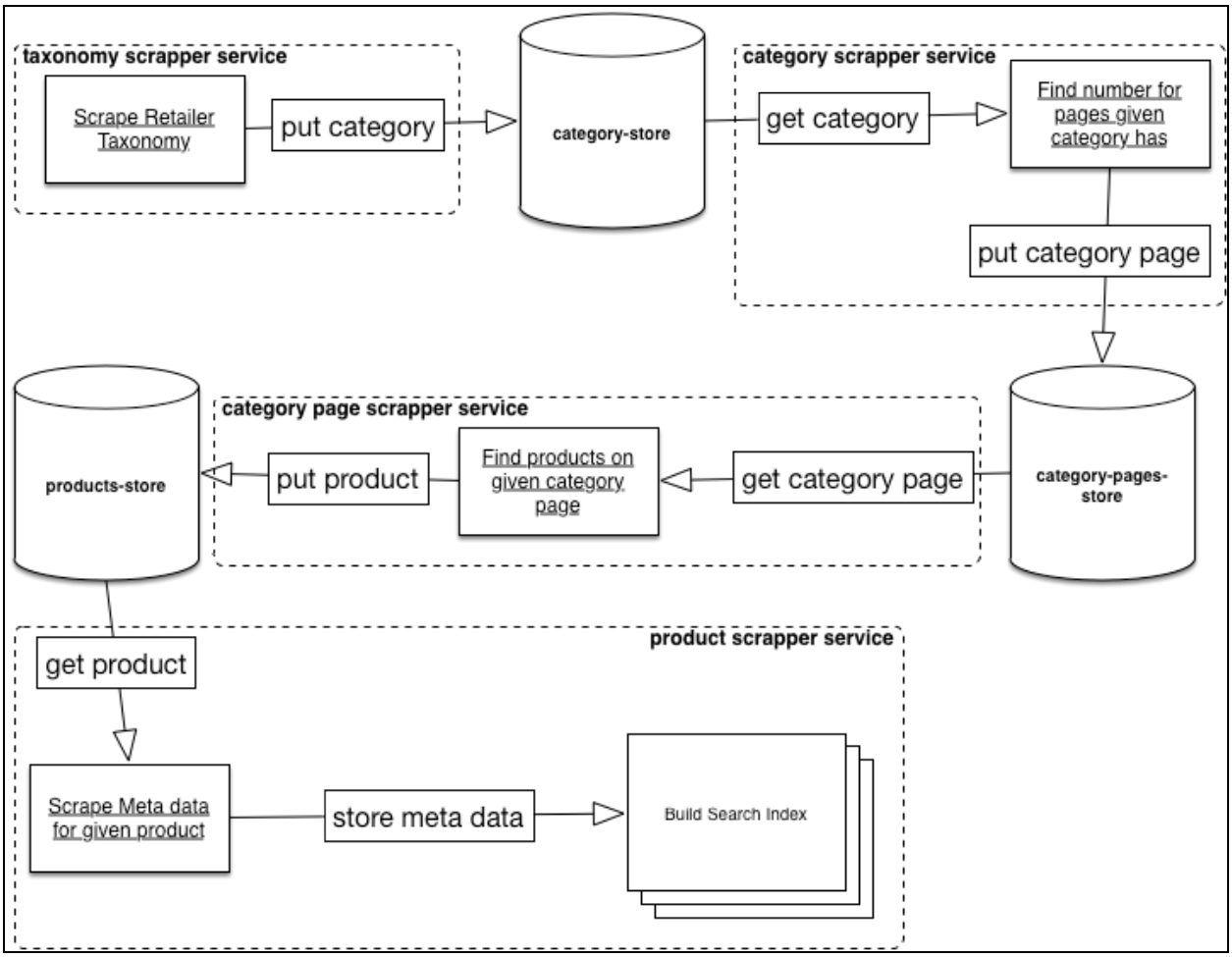

Figure 5: Services Pipeline

This is a form of parallelism through message passing. The outputs of a service are written onto a shared data store with a timestamp. The next service consumes the message from the data store in order of time. The data store can be thought of as a job priority queue.

This architecture has built-in concurrency and can be scaled up in an embarrassingly parallel way. Adding more instances of a service will allow the service to scale up linearly. This type of architecture where one service feeds the next is also known as *collection pipelining* [21]. It has been made popular by UNIX pipes and functional programming patterns like Map-Reduce.

A significant advantage of this architecture is resilience. Failure in one service will not cause the whole application to fail. It will cause the downstream services to finish processing the existing jobs in the pipeline; and they will simply wait for the upstream service to come back online and fill the queue again. Services can be individually taken down, repaired and re-deployed without complex orchestration.

3.2 Services

The process of retrieving and cleaning data is split into four services. Each service is capable of processing a single type of page. They have no knowledge of the other services and can all work independently of each other. Each service is stateless.

Each service fetches a piece of data from a retailer, enriches it and passes it along to another service. Each service is agnostic to the retailer being scrapped. A middleware library acts as an adapter for the retailer being scrapped. This middleware handles parsing of the HTML code and converting it into a structured data the service can operate on. That is, e.g., the webpage showing a particular product will vary drastically between Wal-Mart and Target. The scrapper in

charge of parsing that blob of HTML into a structured list of items is going to have to be aware of the HTML structure of each retailer.

If the resulting structure data format were to be XML, the obvious solution to this problem is using a declarative transformation language like XSLT. There would exist an XSLT style sheet for each type of page for every retailer, which would result in an XML output.

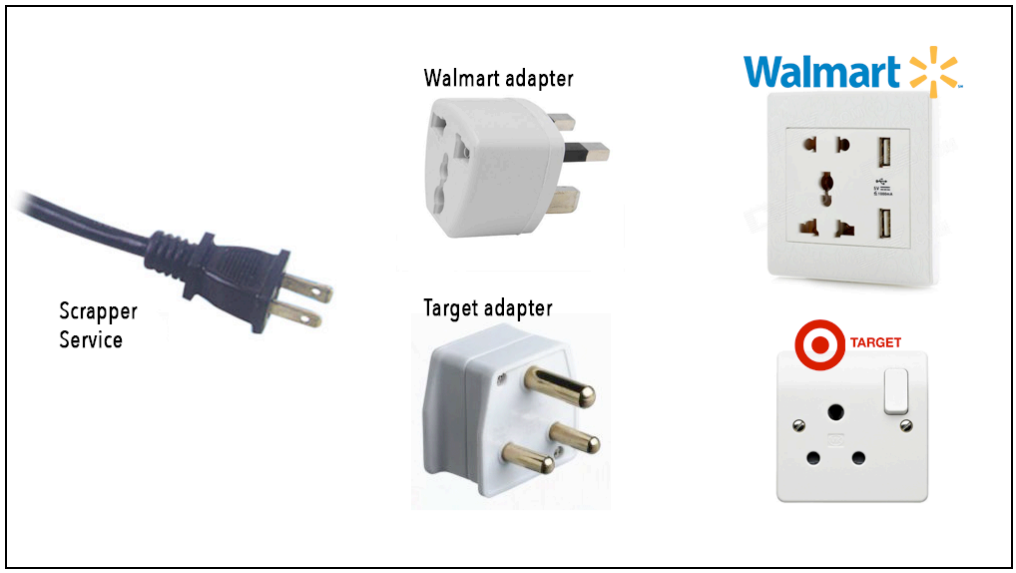

Figure 6: Adapter Middleware Analogy

For this thesis, we have opted to use JSON over XML because it's a superior structured data format. The era of XML-based SOAP is coming to an end and being replaced with JSON-based message passing. JSON is computationally less intensive to parse than XML; it is more human readable and has higher information density while transferring over the wire. Because of these reasons, we have opted to use JSON as the data-interchange format.

However, implementing a declarative language like XSLT specifically for transforming HTML to JSON is not easy. JSLT [22] was a project that was started with this goal, but was abandoned seven years ago because of the inherent complexity of the task.

However, we can implement an imperative middleware script to solve this issue with very minimal complexity. We can leverage existing frameworks that can easily parse through HTML, one such library is JQuery, which is implemented in JavaScript. JQuery has out of the box support for selecting and parsing through HTML. It can select elements via HTML IDs, CSS class hierarchies, field types and can even support regular expression matching [23]. And since its written in JavaScript, it has native JSON support.

So for every retailer, we can write a customized middleware adapter in JavaScript-JQuery. This helps these services to be agnostic to the retailer website they are scrapping. An example script is illustrated in Figure 7.

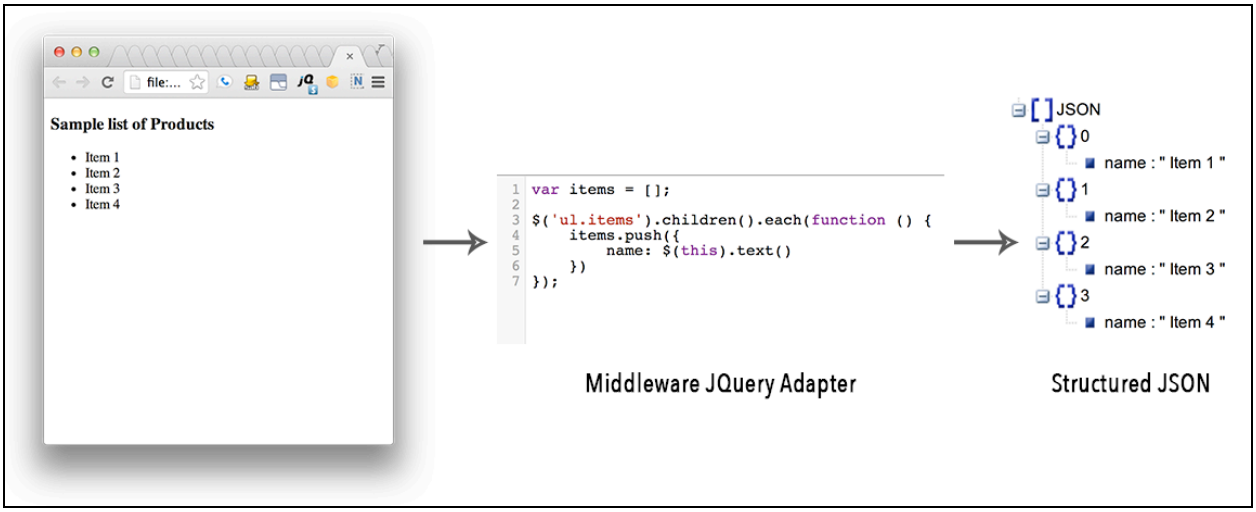

Figure 7: A Simple JQuery adapter to convert a HTML list into structured data

3.2.1 Taxonomy Scrapper Service

The Taxonomy scrapper service is the starting point of the whole system. It is responsible for responsible for building a directed graph of categories from the category page HTML of the retailer. It then visits each leaf node the ontology tree and pushes it into the data store.

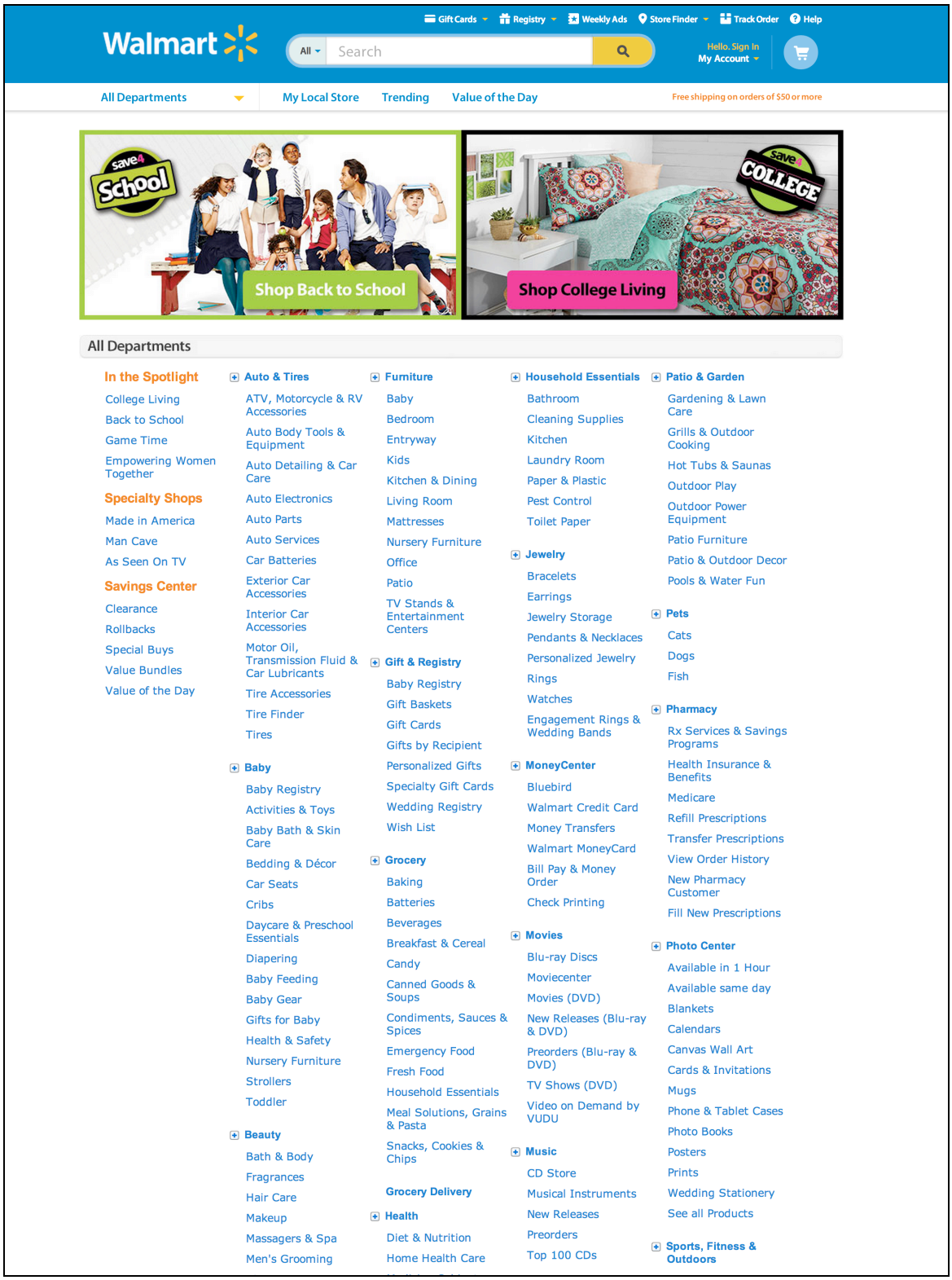
Walmart
Gift Cards
Registry
Weekly Ads
Store Finder
Track Order
Help
All
Search
Hello. Sign In
My Account
All Departments
My Local Store
Trending
Value of the Day
Free shipping on orders of $50 or more
Save4 School
Shop Back to School
Save4 COLLEGE
Shop College Living
All Departments
In the Spotlight
College Living
Back to School
Game Time
Empowering Women Together
Specialty Shops
Made in America
Man Cave
As Seen On TV
Savings Center
Clearance
Rollbacks
Special Buys
Value Bundles
Value of the Day
Auto & Tires
ATV, Motorcycle & RV Accessories
Auto Body Tools & Equipment
Auto Detailing & Car Care
Auto Electronics
Auto Parts
Auto Services
Car Batteries
Exterior Car Accessories
Interior Car Accessories
Motor Oil, Transmission Fluid & Car Lubricants
Tire Accessories
Tire Finder
Tires
Baby
Baby Registry
Activities & Toys
Baby Bath & Skin Care
Bedding & Décor
Car Seats
Cribs
Daycare & Preschool Essentials
Diapering
Baby Feeding
Baby Gear
Gifts for Baby
Health & Safety
Nursery Furniture
Strollers
Toddler
Beauty
Bath & Body
Fragrances
Hair Care
Makeup
Massagers & Spa
Men's Grooming
Furniture
Baby
Bedroom
Entryway
Kids
Kitchen & Dining
Living Room
Mattresses
Nursery Furniture
Office
Patio
TV Stands & Entertainment Centers
Gift & Registry
Baby Registry
Gift Baskets
Gift Cards
Gifts by Recipient
Personalized Gifts
Specialty Gift Cards
Wedding Registry
Wish List
Grocery
Baking
Batteries
Beverages
Breakfast & Cereal
Candy
Canned Goods & Soups
Condiments, Sauces & Spices
Emergency Food
Fresh Food
Household Essentials
Meal Solutions, Grains & Pasta
Snacks, Cookies & Chips
Grocery Delivery
Health
Diet & Nutrition
Home Health Care
Household Essentials
Bathroom
Cleaning Supplies
Kitchen
Laundry Room
Paper & Plastic
Pest Control
Toilet Paper
Jewelry
Bracelets
Earrings
Jewelry Storage
Pendants & Necklaces
Personalized Jewelry
Rings
Watches
Engagement Rings & Wedding Bands
MoneyCenter
Bluebird
Walmart Credit Card
Money Transfers
Walmart MoneyCard
Bill Pay & Money Order
Check Printing
Movies
Blu-ray Discs
Moviecenter
Movies (DVD)
New Releases (Blu-ray & DVD)
Preorders (Blu-ray & DVD)
TV Shows (DVD)
Video on Demand by VUDU
Music
CD Store
Musical Instruments
New Releases
Preorders
Top 100 CDs
Patio & Garden
Gardening & Lawn Care
Grills & Outdoor Cooking
Hot Tubs & Saunas
Outdoor Play
Outdoor Power Equipment
Patio Furniture
Patio & Outdoor Decor
Pools & Water Fun
Pets
Cats
Dogs
Fish
Pharmacy
Rx Services & Savings Programs
Health Insurance & Benefits
Medicare
Refill Prescriptions
Transfer Prescriptions
View Order History
New Pharmacy Customer
Fill New Prescriptions
Photo Center
Available in 1 Hour
Available same day
Blankets
Calendars
Canvas Wall Art
Cards & Invitations
Mugs
Phone & Tablet Cases
Photo Books
Posters
Prints
Wedding Stationery
See all Products
Sports, Fitness & Outdoors

Figure 8: Input of the Taxonomy Scrapper: The Wal-Mart Category Page[3]

---

[3] Courtesy http://www.walmart.com/cp/All-Departments/121828?povid=P1171-C1110.2784+1455.2776+1115.2956-L437

Each retailer has a URL where they display a list of categories and subcategories. Wal-Mart's page is at http://www.walmart.com/cp/All-Departments/121828?povid=P1171-C1110.2784+1455.2776+1115.2956-L437 and Target's is at http://www.target.com/np/more/-/N-5xsxf#?lnk=gnav_more_15_0. These pages are primarily designed as sitemaps, to give the customers a way to quickly navigate through their catalog. These pages are assumed to contain a comprehensive, hierarchized collection of categories under which the retailer has inventory. We can crawl this page, scrape the contents and build a taxonomy tree of the categories. We can now do a breadth first traversal of tree and add every node to the work queue of the next service. The actual job contains the category name, category URL and a timestamp. A breadth first traversal is done instead of depth first is because it is better to have a sample of products from every category rather than an deep dataset from a small part of the graph.

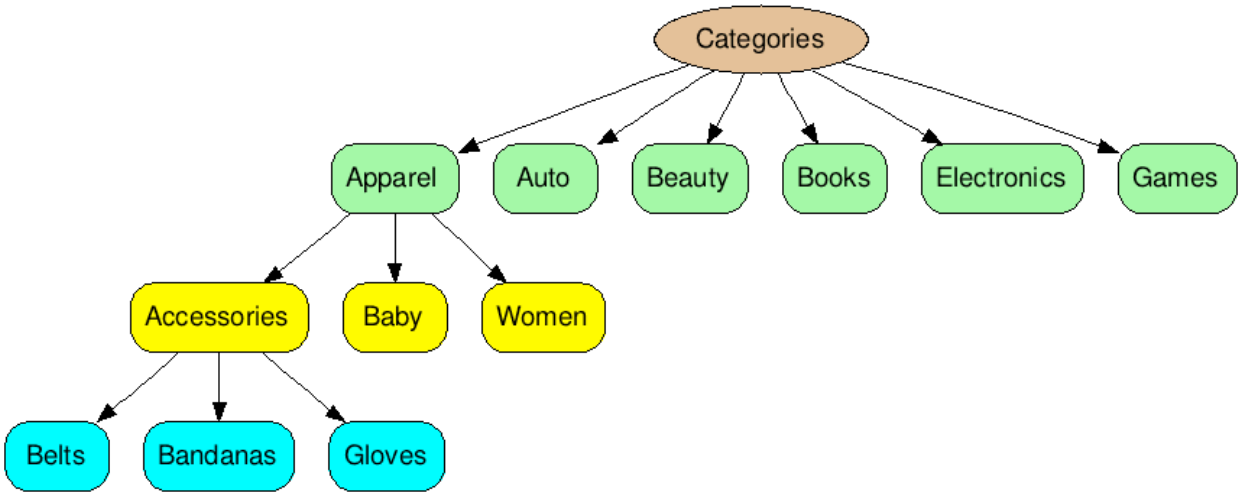

Figure 9: Sample subset of the output by the Taxonomy Scrapper

3.2.2 Category Scrapper Service

This service consumes from the job queue mentioned in the previous section. Each job contains a category name and URL. This service visits the category page and counts how many pages of products it has filed under it. It extracts this data from the pagination section of the page.

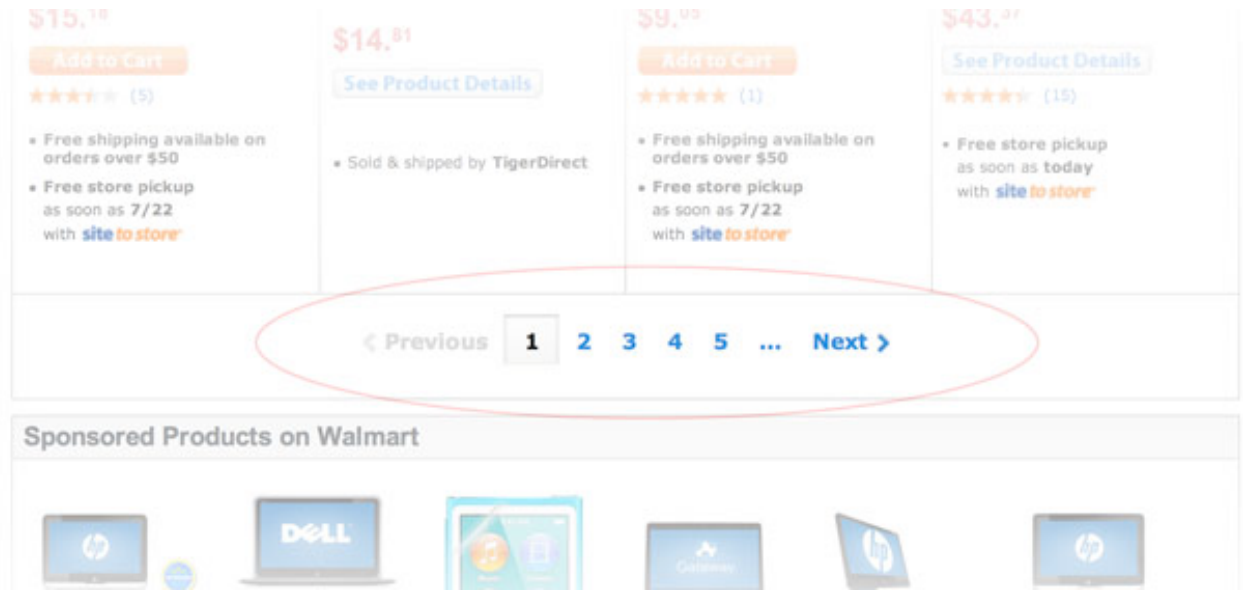

Figure 10: Category Pages to be extracted

Our end goal is to return a list of all the products listed under a category. Each category might have several hundreds of products listed under it. To help the users wade through this list, these items are almost always "paginated." According to Wikipedia, *pagination* is defined as "to divide returned data and display it on multiple pages" [24]. Each of these pages usually contains 10-50 items. The task of this service is to disassociate a given category into its sub-pages and based on the number of pages, estimate how many products each category would contain. Each page of each category is put on the next queue as a job. The meta data includes category URL, category page number and a timestamp.

```
{
  "items": [
    {
      name: "Cirago Lightining SYnc Charge Cable",
      url: "http://walmart.com/product/55412",
      category: "electronics",
      page: "3"
    },
    {
      name: "Ematic 8-in-1 Accessory Kit with Portable...",
      url: "http://walmart.com/product/55414",
      category: "electronics",
      page: "3"
    },
    {
      name: "Belkin Car Auxillary Cable, 3",
      url: "http://walmart.com/product/55415",
      category: "electronics",
      page: "3"
    },
    {
      name: "Ematic 3-in-1 Accessory Kit for iPad, iPhone",
      url: "http://walmart.com/product/55416",
      category: "electronics",
      page: "3"
    }
  ]
}
```


Figure 11: Structured JSON of Items extracted from Category Page

3.2.3 Category Page Scrapper Service

This service gets a category page URL from the job queue and extracts all the products in that particular page. This service takes in a category page URL as input. Every product listed on that page is extracted and pushed on to the next queue. It is augmented with some metadata such as the category and page number the product was extracted from.

This page has summary details about a product, e.g., the title, description and price. This scrapper takes the item and puts it on the final scrapper for a detailer scrape. Refer Figure 11.

3.2.4 Product Scrapper Service

This service is responsible for actually scrapping the product data from the individual product's page. The scrapper will collect the product's name, description and unique retailer id. Extra meta data is also collected. It is unique for every item. For example, for a TV, it might be "dimensions" and "contrast ratio," while for books it might be "author name" and "ISBN". This data is then put onto a processing to be indexed for quick searching in the future. A sample data structure is shown in Figure 12. This data is then passed on to be indexed as described in section 3.3.7.

```
{
  name: "Ultra-Last ULKEBT086B Motorola GMRS/FRS Replacement Battery",
  id: 24727274,
  price: 9.94,
  reviews: null,
  rating: 3.4,
  image: "http://i.walmartimages.com/i/p/00/07/60/97/96/0007609796154_300X300.jpg",
  specs: {
    Battery Type: "Nickel Metal Hydride",
    Multi Pack Indicator: "No",
    Model No.: "ULKEBT086B",
    Shipping Weight (in pounds): 0.25,
    Product in Inches (L x W x H): "5.75 x 3.0 x 1.0"
    Walmart No.: 551505970
  },
  description: "Motorola GMRS/FRS Replacement Battery KEBT086 helps your gadget stay up
    and be on the go!"
  categories: [
    {
      id: 164120,
      name: "2-Way Radios",
      path: "Electronics/Phones & Accessories/2-Way Radios",
      page: 1,
      url: "http://www.walmart.com/cp/164120?showAll=true&bti=0"
    }
  ]
}
```


Figure 12: JSON Data Structure of a 2-way radio

3.3 Implementation Details

The previous section dealt with a high level architecture. The following deals with actual implementation patterns. We will use the example of scrapping Wal-Mart to illustrate the topics.

3.3.1 Language Selection

After evaluating the pros and cons of several languages, Node.js was selected as the implementation language. Node.js is a platform library that allows JavaScript execution outside of the web browser [25]. It is built on the highly optimized Google Chrome JavaScript engine dubbed V8. This allows us to use JavaScript, the same language powering the front-end, on the server side as well.

Node.js met our needs because it has native support for JSON, which we had selected as the default data-interchange format (see section 3.2). Unlike procedural scripting languages like Python or Perl, Node.js is event-driven. Many of the operations in this project are network calls. Event driven programming allows the program to not be blocked when waiting on a request to complete. Several dozen requests can be made concurrently without halting execution of the whole program. When a request is complete, a callback is signaled [26, 27, 28]. This inherent asynchronous nature of Node.js makes it a good choice for high throughput and low latency applications like web crawlers.

The selection of Node.js (which uses JavaScript) to do large scale crawling might seem counter intuitive. JavaScript is after all an interpreted language, and a compiled language like Java or .NET should theoretically outperform JavaScript. This might be the case for computationally intensive problems, but for I/O and network heavy applications, Node.js is the clear winner. This can be seen in the results of the C10K problem. The C10K problem is:

> the problem of optimizing network sockets to handle a large number of clients at the same time. The name C10k is a numeronym for concurrently handling ten thousand connections" [29].

Node.js was able to handle 10,000 concurrent network connections while a compiled .NET application running on Microsoft's IIS server failed the test. On a similar test to compare it with Java, Node.js has better response times and was able to handle more connections [30].

3.3.2 Scaling

The sheer amount of data that has to be processed is staggering. Wal-Mart Supercenters average at 187,000 square feet and house 142,000 different items [31]. But, Walmart.com is not constrained by area. It contains around two million unique items. Each category page only

paginates 50 items at a time. So, that's roughly 4,0000 category pages + 2M item pages + 3,000 unique categories = a minimum of 2,043,000 unique pages to crawl. And this number could easily be much higher because a single item can be in multiple categories.

To accommodate this, the deep mining architecture will have to be able to scale both vertically and horizontally. Vertical scalability is the ability to use all the resources within one node if its specification increases. This is usually implemented using threads. Each thread can be run on a separate CPU for maximum utilization of resources. Horizontal scalability is being able to scale across several nodes, which might be geographically dispersed. This process involves running another instance of the software on another server and load balancing between the two instances.

The service oriented pipelining architecture described in this thesis is inherently both horizontally and vertically scalable. The shared data store acts like a job queue. Each service can be thought of like a single autonomous worker. It pops the latest job from the queue, processes it and then pushes the results onto another queue. It is now apparent that this architecture is embarrassingly parallel. Simply adding a new instance of the application would double the workers and the jobs can be consumed at twice the speed. There is no complex load balancing or cluster management.

Node.js does not however support thread level parallelism. This was a conscious decision by the architects of the language to keep complexity to a minimum. It has a single event loop and achieves concurrency through callbacks [25]. So to scale it vertically, we have to resort to launching a new process. Each process is automatically scheduled by the operating system to run on different CPUs to maximize performance.

To test the optimal number of processes to use for best performance, we benchmarked running several processes on a 2.3Ghz Dual Core Sandy Bridge Intel i3. The results are below:

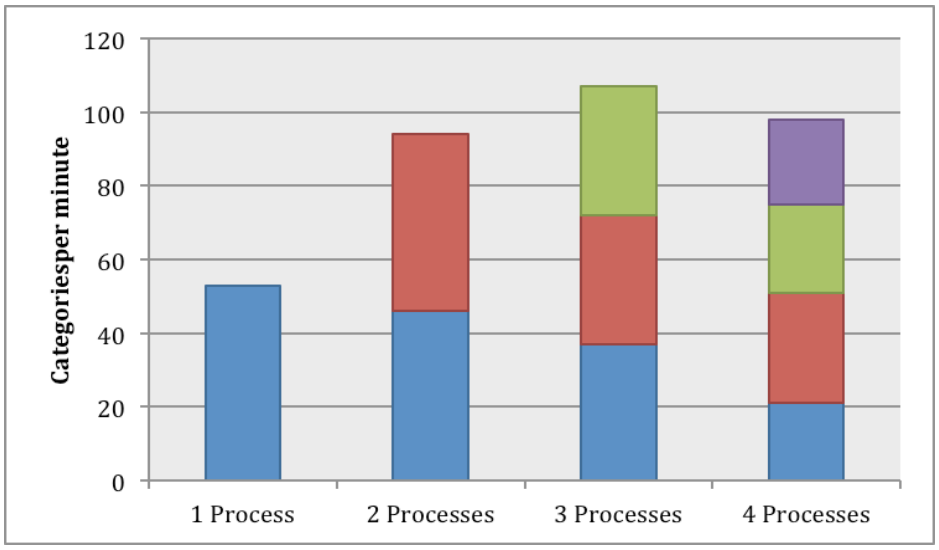

Figure 13: Process level Parallelism: Category Scrapper Service

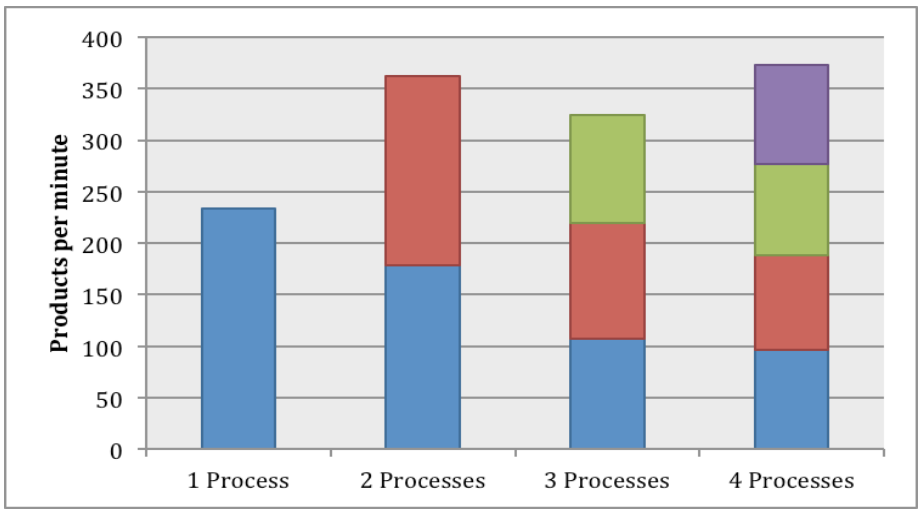

Figure 14: Process level Parallelism: Product Scrapper Service

As we can see, increasing the number of processes increases the throughput, but only up to a point. Beyond a threshold, the performance stays the same, or even deteriorates. This can be attributed to a few things: fight for CPU cycles, local network connection throttling, or remote network throttling. This finding is in line with the graph hypothesized in the Universal Scalability Law [32]. The throughput for the Product scrapper service is much higher than the category scrapper in terms of records processed per minute because the middleware transformation logic is much smaller and simpler.

To study the horizontal scalability of this system, we needed to run it on several different machines. For this test, we ran our instances on disposable virtual servers in the cloud, specifically Amazon Web Services. These cloud servers save time and labor commitments of managing physical machines, especially since we are going to be deploying upwards of 12 different virtual servers. For this particular benchmark, we used a 't2.micro' Amazon EC2 node. This has the performance roughly of a single core 2.5Ghz Low to Moderate Intel Xeon family processor [33].

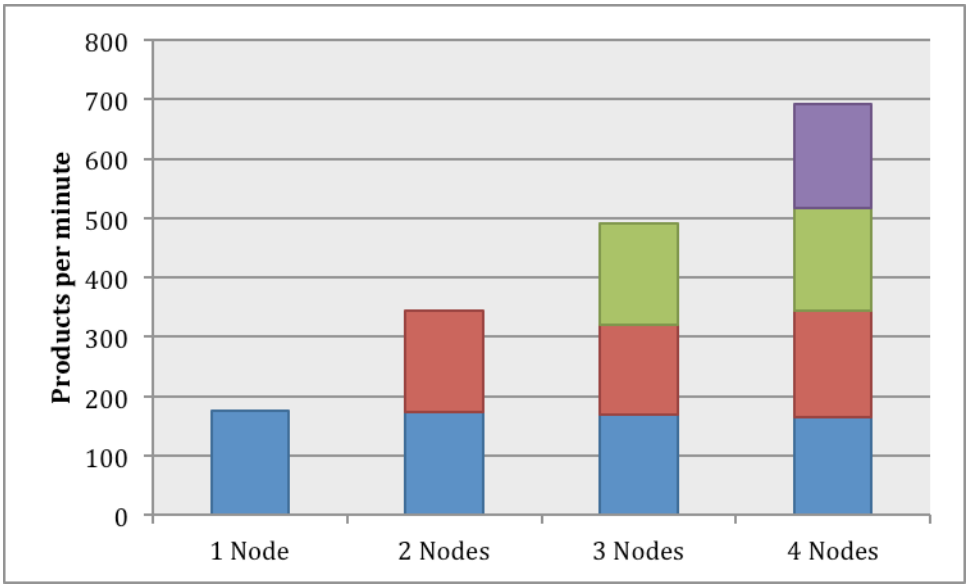

Figure 15: Node Level Parallelism: Product Scraper Service[4]

We can see that the scalability increases linearly. Since they don't share CPU cycles or the same network interfaces, each node has more or less the same throughput. This allows us to scale up our operations linearly and predictably simply by adding more servers.

To do the actual aggregation of data, a series of 3 separate nodes were used. Only one process was spawned on each machine. Each node was able to process on average about 120 pages per minute (both category and product pages). We were able to completely rebuild Wal-Mart's catalog of 1.92M unique items along with their ontology in 63.9 hours. Target's product catalog is significantly smaller than Wal-Mart, this is because Wal-Mart has a "marketplace" which allow third party suppliers to sell through Walmart.com. Target's catalog of 352,000 items and ontology were mined in approximately 5 hours. Targets website design is less bloated and

[4] The per node throughput is slightly less than Figure 14, because the virtual server used to run these tests are slower and have one fewer CPU core.

has significantly less resources that have to be downloaded. This accounts for the significantly reduced retrieval time. This time can easy be cut down if more servers were purchased. It's a tradeoff between cost and speed.

3.3.3 DDOS Blacklist

The rapid fire requesting of resources on Wal-Mart.com triggered an automatic blocking of IP addresses of our servers by their system. This sudden influx of traffic from a small IP range set off flags in their system that detected us (falsely) as an attempted a distributed denial of service attack. They promptly shut us off completely and halted all mining operations. Wal-Mart has this protection in place because each request requires the page to be dynamically constructed from a database and is taxing on their backend servers.

Upon further investigation, we learned that Wal-Mart had a whitelist of search crawlers that they allow to crawl their site. This allows them to automatically block denial of service attempts without blocking search bots. It seemed very unlikely they would add us to their whitelist, so we had to come up with creative solution around the restriction.

ProxyRack (http://www.proxyrack.com) is a proxy as a service company that allows purchasing IP addresses in bulk at nominal fees. We were able to lease 100 different IP for a period of one week. We were then able to write middleware libraries to our scrappers, which switch between these IP addresses every 500 requests. There was a slight increase in latency and reduction in the scraping speed added by the proxy.

3.3.4 Job Queue

The data store that exists between the services acts as a priority queue. The services consume from this queue. This data store has to be high concurrency and low latency. This piece of software is one of the cornerstones of the whole architecture. It has to be able to handle requests thousands of times per second from at least a dozen different consumers spread out geographically. Any lapse in durability of this queue will cause data loss and halt the data mining process.

We chose to use Redis (http://redis.io/) as the data store to back our priority queue. It is highly optimized open source software that can handle up to 5,000 connections and 1,000 storage operations per second [34]. It can achieve these speeds by doing all the processing and storing in-memory. Changes are persisted to disk at regular intervals for durability.

This means that the servers, which host the Redis, must have enough memory to hold the inputs, outputs and intermediary data for the whole scrapping process. This includes raw HTML blobs of each product and category page, structured data and search indices. In the first run, the server was hosted on an Amazon EC2 c3.2xlarge instance that offers 15 GB of ram. The entire ram was consumed within the first 24 hours. Once there is no free memory left, the mining process halted. We had to expand to a larger c3.4xlarge instance that offered 30 GB of memory to be able to finish the mining process. These instances are expensive, costing almost $ 58.25 for the 64 hours of usage. Similar to the topic discussed in Section 3.3.2, this was a tradeoff between cost and speed.

3.3.5 Handling Duplicates

Wal-Mart's ontology graph is cyclical and exhibits multiple-parenthood. One category or item can be referenced from multiple categories. Products are duplicated throughout the site. It is important that we do not waste precious CPU cycles and memory on duplicates. We needed a fast in-memory way to check if we had already scraped an item.

We used a bloom filter to solve this issue. Bloom filters are probabilistic data structures, which can allow us to test with certainty if a product or category has not been scrapped yet. Bloom filters use one or a series of hash functions to determine if the element exists in a set. It returns false if the element is known 100% the element is not in the set, and returns true if the element is probably in the set [36].

The complexity depends on the hash function being used. It has best case performance of O(1) and the worst case of O(N) [35]. We use the DaBlooms implementation of the bloom filter. It is written in pure C and is highly optimized. See https://github.com/bitly/dablooms.

This is implemented as a network service, which returns a Boolean for every category or product ID. In our benchmarks, this service was very fast, with the network latency accounting for most of the time.

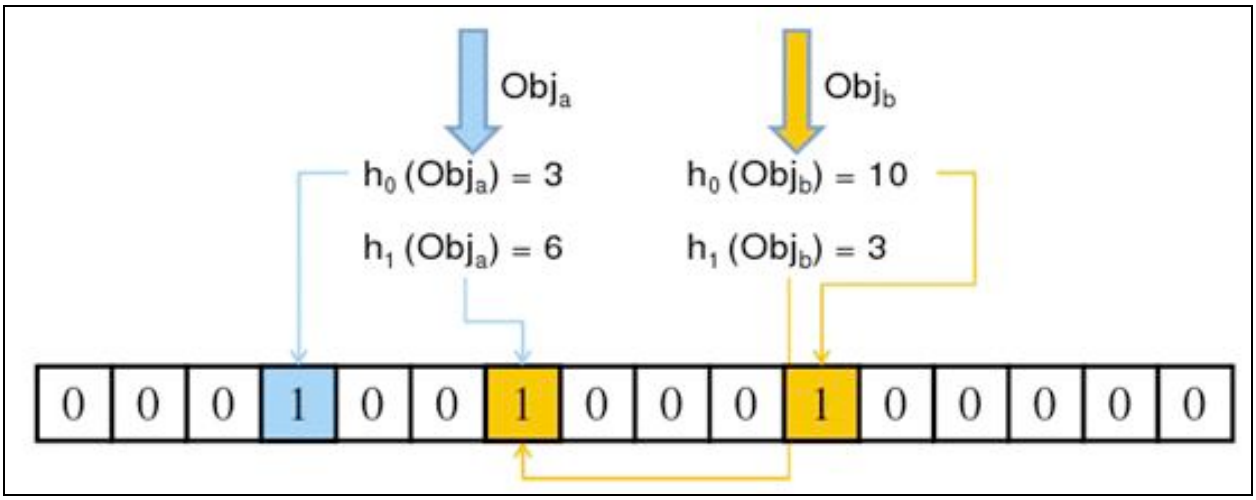

Figure 16: Bloom filter in action[5]

3.3.6 Storage

The final product data needs to be persisted onto disk for future querying and use. This was originally done with a MySQL database. Unfortunately, there were several complications with this data store that caused us to switch to the NoSQL database MongoDB.

Each product scrapped had different associated meta data. For example, a TV might have meta data about the dimensions of the TV, pixel density and contrast levels, while a shirt might just have a single field for size "S/M/L/XL". This was accommodated for by having a separate "Products Metadata" table. It had a one-to-many relationship from the "Products" table. This table grew several orders of magnitude faster than the products table.

The tipping point of complexity happened when the SQL queries to store data were taking too long to complete. Each write and read operation required joining data between three

[5] Courtesy http://loveharbor.net/?tag=bloom-filter

separate tables. MySQL prevents race condition by using a blocking save approach. Only one operation is performed at a time. This was creating a bottleneck in the scrapping process.

We switched to a NoSQL database, MongoDB. It sacrifices the ACID properties of MySQL for fast write speeds. It was able to achieve close to 2,000 writes/second on an EC2 t2.micro instance, while MySQL was able to achieve about 550 writes/second. MongoDB achieves this by keeping all the data in memory (similar to the priority queue discussed in section 3.3.2). It is optimized for write performance; all put requests are non-blocking and asynchronous.

MongoDB is a document-oriented store, which is completely schema-less [37]. This means that data that needs to be stored does not need to be structured around a pre-determined format. Raw JSON data can be pushed onto it from the services without any preprocessing. This alleviates the complexities that were present in dealing with MySQL

3.3.7 Search

Initially we just used MySQL fuzzy queries for searching through the dataset. It was functional, but it was neither efficient nor fast. MongoDB does not have a fast and efficient search engine built in either. So we had to turn to other solutions.

ElasticSearch is a "powerful open source search and analytics engine that makes data easy to explore." [38]. It has power tokenizers and word stemmers (borrowed from the Apache Lucene library) which allow it to intelligently correct spelling errors and provide more accurate results. Similar to MongoDB, ElasticSearch is completely schema-less. Large JSON blobs can simply be dumped onto ElasticSearch, and it will build a reverse index of all the fields present. It also has fast faceting, which allows us to filter products by their metadata. Search queries similar to SQL can be written to search the dataset. A GUI query builder is bundled to make this easier.

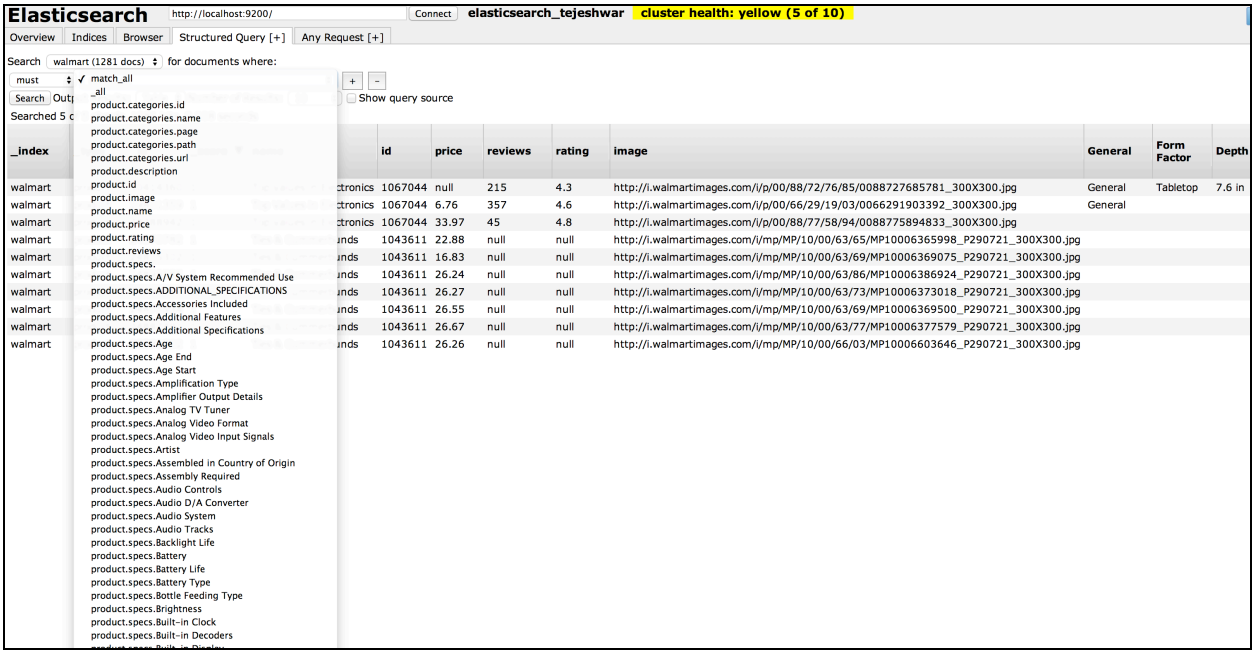

Figure 17: ElasticSearch's Powerful Query Builder

3. 4 Testing

Testing distributed applications is not always an easy task. In the case of the proposed architecture, however, the system is fairly straightforward to test. Since each service is discrete and has no dependencies, unit tests can be written to test and verify the functionality of each service. In addition, queue statistics can be monitored to make sure queues are reasonable in size and when all queues are empty, the crawl is completed.

Most of the problems in distributed systems similar to this usually arise at scale; unit testing would not able to catch those errors. We were able to catch and fix these errors by having a special "error" queue for each service. When a job fails, the details of the failure along with a

full stack trace are pushed onto this queue for further analysis. The scrapper skips the failed jobs and proceeds with the next jobs in queue.

3. 5 Results

Measured results from the traversal and time taken are shown in the following table:

	Wal-Mart	Target
Unique Items	1.92M	352,000
Time Taken	63.9 hours	5 hours
Product Categories	666	334
Hierarchical Connections	786	324

A frontend was built to explore the data aggregated. A full replica of the Wal-Mart and Target Product catalog was downloaded. The Wal-Mart data has been fully indexed can be explored that a GUI web interface at http://walmart.cmyk.io.

Because we are storing all search indices in memory, we need a fairly robust server. Wal-Mart's search indices alone are approximately 30GB in size. This requires an expensive 'm3.xlarge' Amazon EC2 node which costs a little over $200 USD/month. In the interest of reducing costs, only Wal-Mart's indices were built for exploration.

Both Target and Wal-Mart data is analyzed in Chapter 4.

4. ONTOLOGY ANALYSIS

4.1 Selected Ontologies and Analysis Approach

Once we have implemented the software based on the described architectural model, we can start gathering and analyzing ontological data. Since the product ontology involves representing hierarchy between different objects, it can be represented as a graph, and graph algorithms can be applied to learn more. The following section compares and contrasts various facets of these ontologies.

Wal-Mart, Amazon, Target and Kroger were the retailers considered for this study. Since aggregating and storing data from all these retailers is computationally expensive, two of these retailers were selected.

Of the four, Amazon is different from the others because of its absence of physical brick-and-mortar stores. Their product catalog is several times bigger than the others and their ontology is also much deeper and intricate. This is because they have the advantage of not being limited by physical store space.

Wal-Mart and Kroger are the top two retailers in America [39]. Although Wal-Mart's revenue is roughly five times that of Kroger, their grocery-related ontologies are similar and not different enough for an interesting comparison.

Wal-Mart and Target were the top finalists. They both have an online e-commerce website as well as brick and mortar stores, but their ontologies differ. The entire product catalog along with their ontological classifications and meta data were extracted and stored locally. As mentioned in Section 3.3.2, Wal-Mart's catalog of 1.92M unique items was mined in 63.9 hours while Targets 352K catalog was mined in 5hrs.

To analyze these two product ontologies, we used a combination of scientific computing libraries including:

- Matplotlib – 2D plotting library in Python
- NumPy – Numerical and scientific computing library in Python
- GraphViz – Graph visualization for small graphs
- NetworkX – Graph visualization for very large graphs
- Gephi – Graph metrics

4.2 High Level Comparison

The first aspect to compare is the overall structure. To get a 50,000-foot view of how these ontologies look like, we plotted their category structure as a graph using NetworkX.

In Figure 18 and 19, we visualize Wal-Mart's and Target's categories in their entirety. Each node represents a product category and edges between each node represent hyperlinks between the categories. This plot gives us immediate insight into the relationships operating between the categories.

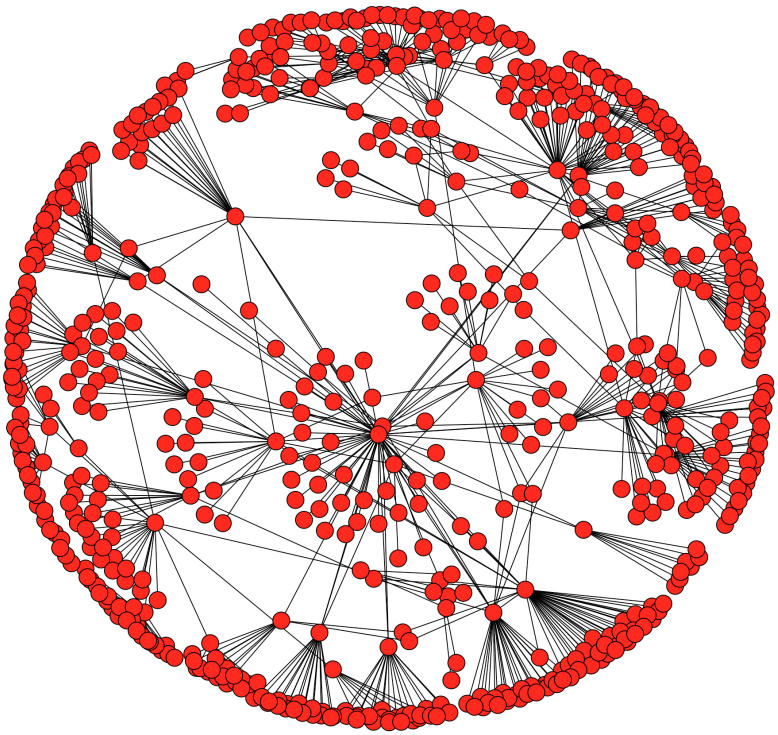

Figure 18: Wal-Mart Ontology: 666 Nodes, 786 Edges

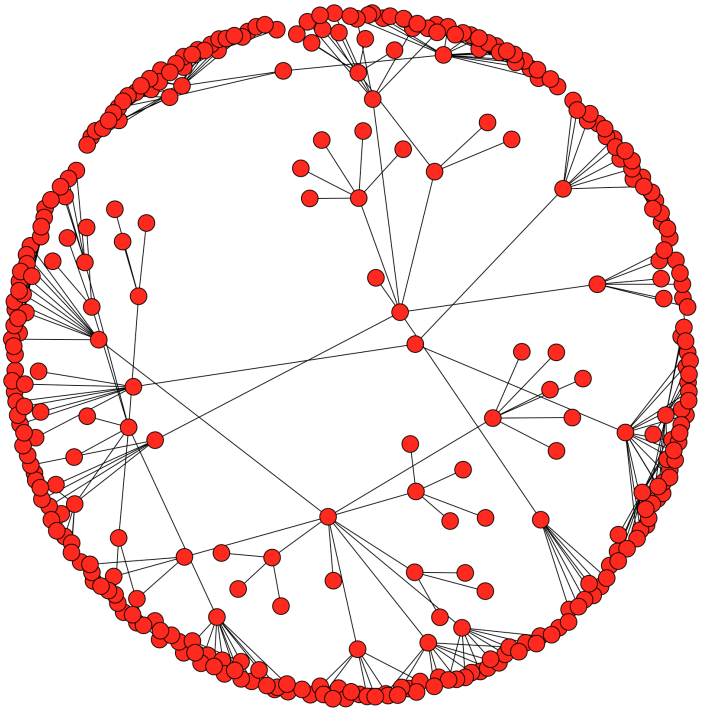

Figure 19: Target's Ontology: 334 Nodes, 324 Edges

There is no “top category” in either the Wal-Mart of Target ontology. It is evident from simply viewing these graphs that Wal-Mart’s ontology is larger, deeper and more intricate than Target’s. Target’s graph is a hierarchical tree, no more than 3 layers deep. Wal-Mart’s, on the other hand, has up to 6 layers of sub-graphs.

The Wal-Mart ontology exhibits multiple-parenthood; each child node can have one or more parents. There are even cyclical relationships within the ontology. For example: ‘Home Theater’ has a hyperlink to ‘Televisions’ and vice versa. The Target graph, on the other hand, is a simple tree structure, with each child item having only one parent.

We continue to explore these graphs in the upcoming sections.

4.3 Structure

From the graphs illustrated in the previous section, we can visually compare their structure and complexities. But, for a more formal measure of complexity of the graph, we can compute the connections per node to its neighbors (degree of the graph) and the average distance between nodes (closeness centrality).

The following are histograms of degree of nodes (both X and Y are on a log scale).

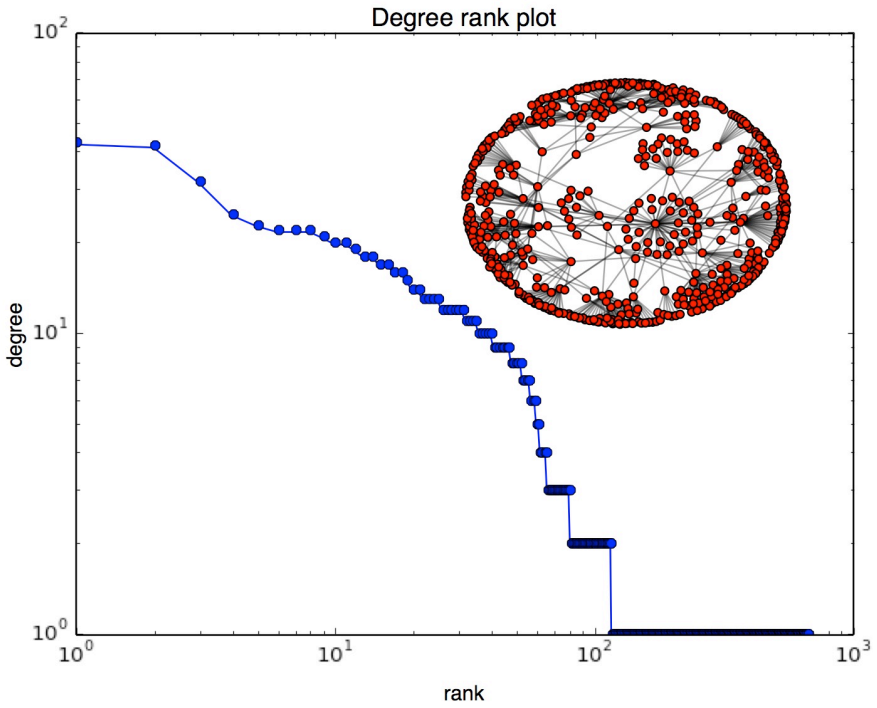

Figure 20: Wal-Mart's Degree Rank

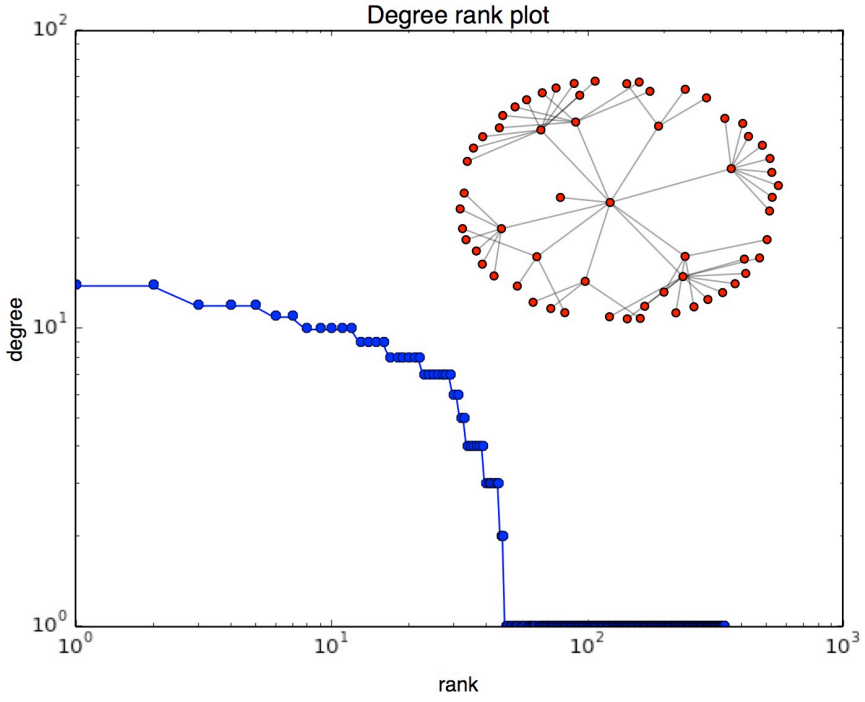

Figure 21: Target's Degree Rank

The histograms show the degree on the Y-Axis and the number of items matching that degree on the X-Axis.

To reduce complexity, the entire graph is treated as undirected. (If it were treated as directed, we would have to deal with a 2-dimentsional degree distribution due to in and out degrees). From the graphs, we can see that the Wal-Mart ontology has a maximum degree of 42 while the Target ontology has a maximum degree of only 13. This is in line with our previous hypothesis of Wal-Mart's ontology being larger and more intricate (more connections, higher ranks).

An interesting observation about both ontologies is how the rank rapidly decays (this is a log scale graph) as the degree increases. Most of the nodes seem to have between 1-5 connections. Only a small percentage of nodes have high degrees.

We can also measure the distance of every node in the graph to every other node using the closeness centrality measure. This value is computed using Gephi's implementation of a technique formulated in Brande's 2001 paper titled "A Faster Algorithm between Centrality" [40].

The results for this computation are graphed in Figure 22 and 23. The X-Axis has the length of the graphs, while the Y-Axis shows the number of nodes which have that length. From these graphs, we can tell that:

- Average path length in Wal-Mart graph: 5.4
- Average path length in Target graph: 3.1

This gives us a measure to compare the relative depths of the graphs. We made a hypothesis in Section 4.2, simply by intuitively studying the visual structure of the graphs. The computed depths confirm that hypothesis that Wal-Mart's ontology is deeper.

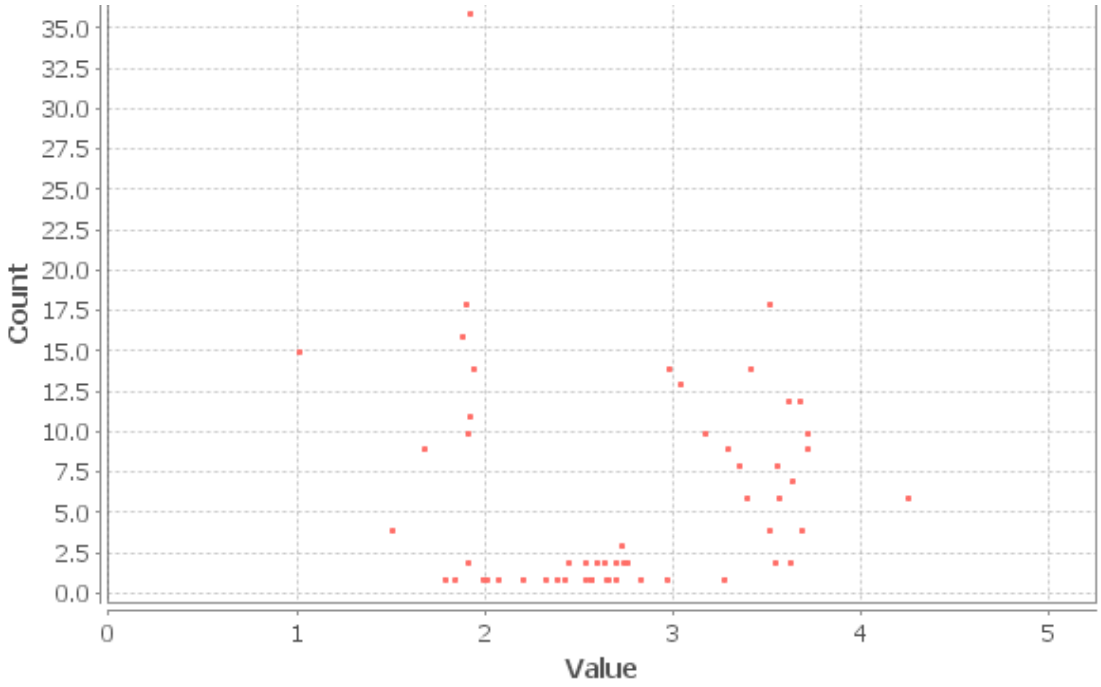

Figure 22: Closeness Centrality Distribution of Target

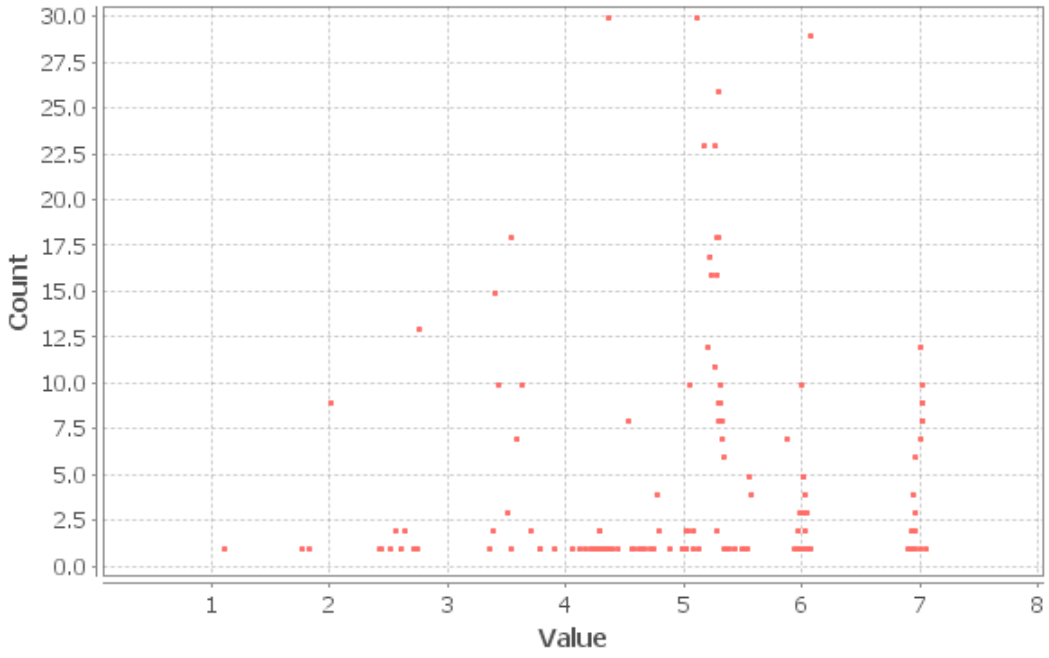

Figure 23: Closeness Centrality Distribution of Wal-Mart

4.4 Clusters

From Figures 18 and 19, we can hypothesize that Wal-Mart's ontology has a higher networked community than Target's. A 'community' in a network refers to a cluster of nodes bunched together because of similarity. To get a clearer representation of the communities and clusters, we have to use a force-directed layout to space out the graph.

We use Gephi's implementation of the Force Atlas 2 engine [41] to help us spread out the nodes and visualize communities.

Wal-Mart's ontology (Figure 24) looks busy. There are more clusters and fewer isolated communities. We can clearly observe the multiple-parenthood in Figure 24. This means that each category or product can have multiple categories as parents. This is an attempt by Wal-Mart to duplicate items across their taxonomy so that people will have an easier time finding them.

Target's ontology (Figure 25) , on the other hand, imposes a strict taxonomy. Cycles do not occur in the graph. Most of the clusters are islanded. Each category or product can have at most one parent.

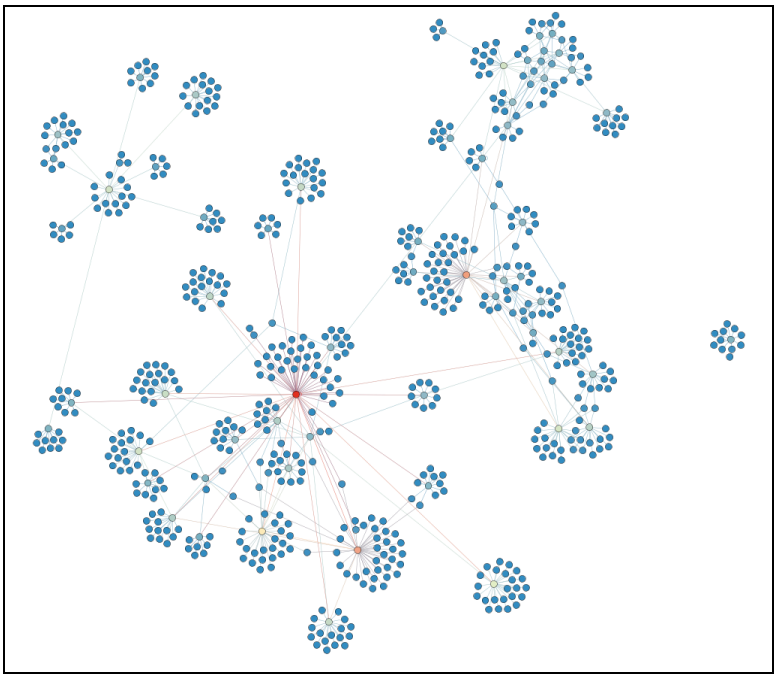

Figure 24: Wal-Mart's Clustered Ontology

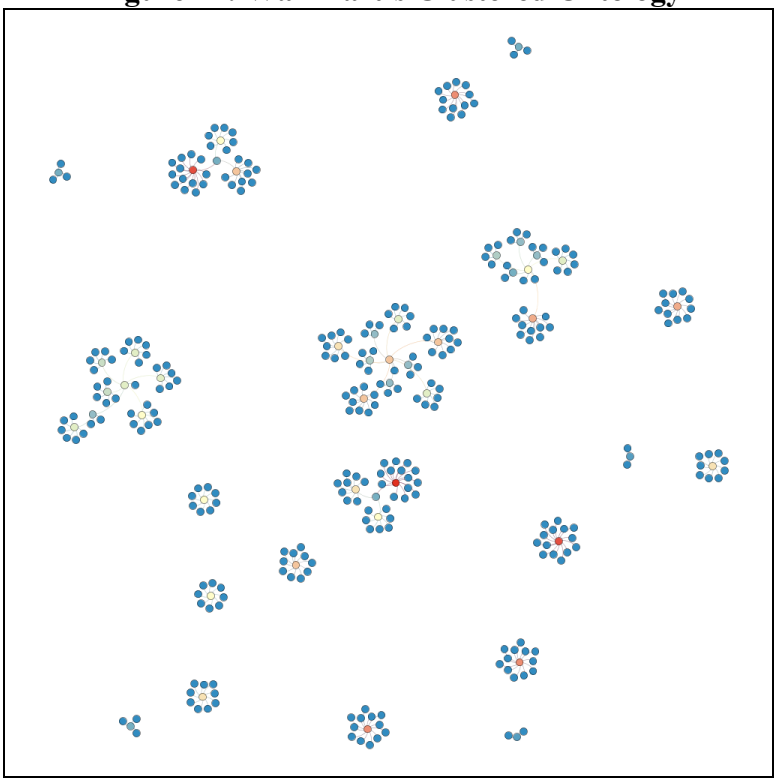

Figure 25: Target's Clustered Ontology

5. CONCLUSIONS

5.1 Summary

This thesis has covered several topics rapidly building large, real world ontologies that can pave the road to the smart semantic world of tomorrow.

In section 1.3 we specified the goals of this thesis. The primary aim was to describe a scalable architecture to aggregate product data and automatically build the product ontology from retailer websites. This is an architecture that allows parallelism without needing to worry about deadlocks or race conditions. In section 3, we described in detail the architecture and implementation details.

A second aim was to actually implement this architecture and have it aggregate data on a few retailers. This was also achieved. The software was implemented and was used to aggregate data from Wal-Mart and Target. Appendix A, B, C and D provide code fragments which are central to the implementation. A fully indexed and searchable replica of the Wal-Mart catalog can be found at http://walmart.cmyk.io.

The third aim was to study how to aggregate product data. In section 4, we viewed the aggregated data, structured the taxonomy as a graph and applied graph theory algorithms to formally compare and contrast the ontological differences between the two retailers.

We were able to meet the required criteria for coverage and efficiency. Coverage can be broken down into Semantic Coverage and Completeness of Mapping. In terms of Semantic Coverage, the crawlers were able to do a breath first discovery of all the products in the retailer's catalog. We captured all of the meta-data regarding entities, attributes and relationships available on the retailer's website. In terms of Completeness, we captured all of the entity, attribute, and

relationship instance data and successfully created a 1:1 mapping of the entire dataset for each retailer. Thus, our traversal of the retail sites was met the completeness criteria.

Efficiency can be further divided into Traversal Efficiency and Query Access Efficiency. Traversal Efficiency is the time to traverse the retail websites to create the product database. We were able to build an architecture that (although IO bound) can scale linearly. The discrepancy between the crawl times between Wal-Mart and Target (see table in section 3.5) might seem contrary to this statement, but the difference was not a caused as a result of our architecture. Rather, the difference can be attributed to Target's mobile website, which is faster to access and simpler to access. Regarding Query Access Efficiency, which is the time to access products in the ontology we built, based on benchmarks, querying the product database that was re-built locally was significantly faster than the native implementations available at either of the retailers.

5.2 Potential Impact

This project is just one of the steps needed to build the infrastructure that will be needed for computers to become situation aware in a smart semantic world.

At present, there does not exist a publically available product ontology containing information from major retailers. We believe we have crossed the "structure chasm" [42] by providing an automated way to build such large structured ontologies of products quickly.

Each product in this database has extensive metadata of descriptive attributes. This will allow future applications to query information in the database based on hierarchy and characteristics rather than just text. We are hoping the results of this thesis have broad-reaching implications in building the smart semantic world of tomorrow.

5.3 Future Work

The results of this thesis provide foundational data to build complex applications in the fields discussed in Chapter 2: ontologies and knowledge representation, Internet of things, and smart semantic worlds. Future work can be classified into short-term goals within the next year and long-term goals within the next 2-5 years.

5.3.1 Short Term Goals

5.3.1.1 Super Ontology

This thesis maps out the product catalogs of different retailers. But, to make this data usable in building smart objects, these retailer-specific ontologies would have to exist inside a common ontology. We are now faced with the challenge of analyzing different ontologies of different retailers and merging them into a super ontology.

There may be some parts of the ontologies that could never be correlated because of an ontological mismatch. But, we believe that by associating these ontologies with the WordNet[6] hierarchy, we can come up with a more useful ontology that encompasses not just the word concepts that WordNet provides but also reaches down below the level of English worlds to identify product classes that humans recognize and use in their everyday lives.

[6] WordNet is an open lexical database developed at Princeton University – see http://wordnet.princeton.edu/

5.3.1.2 Price Comparison and Product recommendation

The thesis provides a completely indexed catalog of all products. This can be used to build a comparative shopping application that suggests the cheapest retailer to buy a particular item.

Since we have extensive metadata on each product, similar products could be clustered with an algorithm like k-nearest neighbor. Product recommendations system have also been built that use the semantics embedded within relationships and Bayesian networks to make accurate product recommendations [43].

This mined data is very volatile. Prices fluctuate, new items are added and old items are removed on a daily basis. To remain current, this data has to be refreshed often. This volatile data would require constant monitoring to prevent the dataset from going stale. Some architectural changes would have to be made to make the scrape process real-time.

5.3.1.2 Data Donation and Ontology-as-a-service

We could create a web service that provides ontology-as-a-service over HTTP. A server, which responds to requests on product catalog and taxonomical relations, would be a boon to future developers. As far as the developer is concerned, this server would be a black box; they can start utilizing the data without having to be aware of implementation details, architectural choices or *big data* issues dealt with in this thesis. This data could also be provided in different formats: JSON, XML and big data query languages like SPARQL.

This data could be donated to DBpedia. It would add to an already impressive amount of structured data they have already amassed. Copyright ramifications would have to be considered

before disseminating this information. The data in question contains material that may be copyrighted, and the retailers might not want their data to be openly available on the Internet.

5.3.2 Long Term Goals

5.3.2.1 Object Recognition

This thesis provides us with a labeled, structured, and categorized database of product *types* along with their pictures. If we want computers to become situation-aware, so that they recognize and react to the world around them, then one next step would be to correlate object instances in the world around us with this database. Recognizing objects around us in 3D space is a difficult problem as it involves image recognition. Attempts have made to recognize small numbers of simple objects with the Xbox Kinect with good accuracy [44] and large numbers of common objects with poorer accuracy via the Kindle/Amazon Fire Phone [14]. Large scale, accurate implementations are yet to be available.

Virtual Worlds like Second Life we can represent the world around us. The can be used to build queryable mirror worlds in which the objects may be tagged with type and provide easy to use 3D representations. One way to improve accuracy would be to learn the 3D object representation of an object inside a virtual world and correlate that knowledge with items in the real world. This would be a fast way to a way to acquire a large collection 3D data. This brings up questions of representational adequacy – do these virtual world models have the ability to represent real world items accurately?

Advancements in this area would have significant implications in augmented reality. If objects around us could be recognized quickly, they can be overlaid with useful information. This goal is being made a reality by the Kindle/Amazon smart phone discussed in Chapter 2.

5.3.2.2 Situational Awareness

Once computers can accurately detect and tag all the objects around them, in addition to having a detailed ontology of products, they would need to become aware of their 3D positions in space. This knowledge would enable some interesting applications.

Computers these days are GPS-enabled and can easily know where there are (or about where they are), but they cannot yet reason about their environment. Having this environmental data will allow computers to become situationally aware. They would be able to reason and make assertions: for example, I'm *in a place that is surrounded by items, which are listed under kitchen appliances; ergo, I am probably in a kitchen.* Product ontologies are a step toward this situation awareness ability. When computers can use this kind of ontological data to deduce facts about their environment, they will be able to interact with humans and the world around them as first class citizens.

6. REFERENCES

[1] A. Eguchi and C. Thompson, "Towards a Semantic World: Smart Objects in a Virtual World," International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM), Vol. 3, 2011, pp. 905-911.

[2] T. Bittner, M. Donnelly and S.Winter,, "Ontology and semantic interoperability." In *Large- scale 3D Data Integration: Problems and Challenges*, D. Prosperi and S. Zlatanova (Eds) CRCpress, 2005, pp. 139–160.

[3] T. Bannon, "Network Query and Matching System and Method." US Patent 6,963,863 issued on Nov 8 2005.

[4] "Walmart Open API" [online], https://developer.walmartlabs.com/ (Accessed: Jun 2014).

[5] "Aristotle's Categories" [online] http://plato.stanford.edu/entries/aristotle-categories/ (Accessed: Jun 2014).

[6] S. Grimm, P. Hitzler, and A. Abecker, "Knowledge Representation and Ontologies," *Semantic Web Services*, pp. 51–105, Jan. 2007.

[7] A. Pichler and A. Zollner-Weber, "Sharing and debating Wittgenstein by using an Ontology," *Literary and Linguistic Computing*, 2013.

[8] H. Kopetz, 2011. Internet of Things. Springer US, pp.307-323

[9] J. D. Eno and C. Thompson, "Virtual and Real-World Ontology Services," *Internet Computing, IEEE*, vol. 15, no. 5, pp. 46–52, 2011.

[10] W. Holmes and P. Kogut, "DAML" PowerPoint, Lockheed Martin, 2001.

[11] "Everything is Alive" [online] http://vw.ddns.uark.edu/index.php?page=goals (Accessed: Jun 2014).

[12] C. Thompson and F. Hagstrom, "Modeling Healthcare Logistics in a Virtual World," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 100–104, 2008.

[13] "DBPedia" [online] http://wiki.dbpedia.org/UseCases (Accessed: Jun 2014).

[14] "What is Amazon Firefly and how does it work?" [online] http://www.technobuffalo.com/videos/amazon-firefly-what-is-it-and-how-does-it-work/ (Accessed: Jun 2014).

[15] I.Lee, S.Lee, T.Lee, S. Lee, "Practical issues for building a product ontology system" Data Engineering Issues in E-Commerce, 2005. Proceedings. 2005, pp. 16-25.

[16] “Product Classification Systems as Web Ontologies” [online] http://www.ebusiness-unibw.org/ontologies/pcs2owl/ (Accessed Jul 2014)

[17] M. P. Papazoglou and W.J. Heuvel, “Service Oriented Architectures: Approaches, Technologies and Research Issues,” *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 16, no. 3, Jul. 2007.

[18] "Comparison of Business Integration Software" [online], http://en.wikipedia.org/wiki/Comparison_of_business_integration_software (Accessed: Jun 2014).

[19] "Apache Service Mix" [online], http://servicemix.apache.org/ (Accessed: Jun 2014).

[20] "Apache Synapse" [online], http://synapse.apache.org/ (Accessed: Jun 2014).

[21] "Collection Pipelining" [online], http://martinfowler.com/articles/collection-pipeline/ (Accessed: Jul 2014).

[22] "JSLT" [online], http://ajaxian.com/archives/jslt-a-javascript-alternative-to-xslt (Accessed: Jun 2014).

[23] "JQuery" [online], http://api.jquery.com/category/selectors/ (Accessed: Jun 2014).

[24] "Pagination" [online], http://en.wikipedia.org/wiki/Pagination (Accessed: Jun 2014).

[25] “Node.js” [online], http://nodejs.org/

[26] "Release the Kraken: A Story of Node.Js in the Enterprise (PayPal)", Queue, vol 12, iss 2, pp. 8080--8080, 2013.

[27] "Notes from the Battlefield (Node.Js at Walmart)", Queue, vol 12, iss 2, pp. 7070--7070, 2013.

[28] "Why Large Scale Mobile and E-Commerce Apps Use Node.Js", Queue, vol. 12, iss 2, pp. 6060--6060, 2013.

[29] “C10K Problem” [online] http://en.wikipedia.org/wiki/C10k_problem (Accessed: Jun 2014)

[30] “Java vs Node.js” [online] https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/ (Accessed Jun 2014)

[31] "Walmart Statistics" [online] http://news.walmart.com/news-archive/2005/01/07/our-retail-divisions (Accessed Jun 2014)

[32] "Universal Scalability Law" [online] http://www.perfdynamics.com/Manifesto/USLscalability.html (Accessed Jun 2014)

[33] "Amazon EC2 Instance Types" [online] http://aws.amazon.com/ec2/instance-types/ (Accessed Jun 2014)

[34] "Redis Benchmarks" [online] http://redis.io/topics/benchmarks (Accessed Jun 2014)

[35] B. Bloom. "Space/time Trade-offs in Hashing Coding with Allowable Errors," Communications of the ACM, 13(7):422 – 426, July 1970.

[36] "DaBloom Open Source Bloom Filter" [online] http://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting-bloom (Accessed Jun 2014)

[37] "MongoDB" [online] http://www.mongodb.org/ (Accessed Jun 2014)

[38] "ElasticSearch" [online] http://www.elasticsearch.org/ (Accessed Jun 2014)

[39] "Top 10 US Retailers" [online] http://www.ibtimes.com/top-10-us-retailers-amazon-joins-ranks-walmart-kroger-first-time-ever-1618774 (Accessed Jul 2014)

[40] U. Brandes, "A Faster Algorithm for Betweenness Centrality*," *Journal of Mathematical Sociology*, 2001.

[41] V. D. Blondel, J. L. Guillaume, and R. Lambiotte, "Fast Unfolding of Communities in Large Networks," *Journal of Statistical Mechanics*, 2008.

[42] A. Y. Halevy, O. Etzioni, A. H. Doan, Z. G. Ives, and J. Madhavan, "Crossing the Structure Chasm," CIDR, 2003.

[43] J.Lee, H.Chae, C.Kim, K.Kim, "Design of Product Ontology Architecture for Collaborative Enterprises", *Expert Systems with Applications*, Volume 36, Issue 2, Part 1, March 2009, Pages 2300–2309.

[44] A. Eguchi, "Object Recognition based on Shape and Function," BS Honors Thesis, CSCE Department, University of Arkansas, Fayetteville, AR, November, 2011. (PowerPoint)

APPENDIX A – TAXONOMY SCRAPPER

This code fragment is from the middleware data transformer. It traverses through the taxonomy graph and adds each category onto the category queue.

```
/*jslint vars: true */
/*jslint browser: true, node: true*/
/*global angular, $, jQuery, google, alert*/

'use strict';

var winston = require('winston'),
    taxonomy = require('./taxonomy.json'),
    walmartCategories = taxonomy.categories,
    traverse = require('traverse'),
    kue = require('kue'),
    redis = require('redis');

kue.redis.createClient = function() {
    //var client = redis.createClient(6379, '130.184.104.66');
    var client = redis.createClient(6379, '127.0.0.1');
    client.auth('Typewriter39');
    return client;
```

```
};

Array.prototype.contains = function (obj) {
    var i = this.length;
    while (i--) {
        if (this[i] === obj) {
            return true;
        }
    }
    return false;
};

winston.loggers.add('walmart-crawler-stage1', {
    console: {
        level: 'info',
        colorize: 'true',
        label: 'walmart-crawler-stage1',
        prettyPrint: true,
        timestamp: true
    }
});

var print = winston.loggers.get('walmart-crawler-stage1');
```

```
var getCategories = function () {
    // Recursively traverse the walmart taxonomy json and return a list of categories
    var categories = [];
    walmartCategories.forEach(function (item) {
        traverse(item).forEach(function (x) {
            if (this.notLeaf && Object.keys(x).contains('id')) {
                categories.push({
                    id: x.id.split("_").pop(),
                    name: x.name,
                    path: x.path
                });
            }
        });
    });
    return categories;
};

var jobs = kue.createQueue();

var gracefulShutdown = function() {
    print.info("Kill Detected, Waiting for jobs to finish");
    jobs.shutdown(function(err) {
```

```
        print.info("All jobs finished. Ending.");
        process.exit(0);
    }, 10000);
};

process.on('SIGINT', gracefulShutdown);
process.on('SIGTERM', gracefulShutdown);

var stage1 = function () {
    // For each category, push into [QUEUE - CATEGORY]
    var categories = getCategories();

    categories.forEach(function(category) {
        print.info("Pushing to Walmart Category Queue: " + category.name);
        jobs.create('walmart-category', {
            title: category.name,
            id: category.id,
            name: category.name,
            path: category.path
        }).priority('normal').attempts(1).save();
    });

    print.info("Finished pushing " + categories.length +
```

```
            " items into Walmart Category Queue");
}

if (require.main === module) {
    stage1();
}
```

APPENDIX B – CATEGORY SCRAPPER

This code fragment is from the middleware data transformer. It traverses through each category and deduces how many pages there are. Each page is then added onto the category page queue.

```
var getCategoryPages = function (categoryID) {
    var deferred = Q.defer();
    var categoryURL = "http://www.walmart.com/cp/" + categoryID + "?showAll=true";
    print.info(categoryID + " - Requesting: " + categoryURL);

    request(categoryURL, function(error, response, body) {

        if (error || response.statusCode !== 200) {
            print.error(categoryID + " - Request Error : " + error);
            deferred.reject(new Error(categoryID + " - Request Error"));

        } else {
            print.info(categoryID + " - Response From: " + response.request.uri.href);
            var $ = cheerio.load(body);
            var recordCount = $(".SPRecordCount").first().text().replace(/(\r\n|\n|\r)/gm," ");

            //regex to match page and item count
```

```
var recordCountRegex = /(\d+)-\s?(\d+)/; // number-<0 to N spaces>number
var productCountRegex = /(\d+)\stotal/; // number<space>total

//match those regex
var rcMatches = recordCountRegex.exec(recordCount);
var pcMatches = productCountRegex.exec(recordCount);

//if there are matches
if ((rcMatches !== null) &&
   (rcMatches.length === 3) &&
   (pcMatches !== null)) {
   var productsPerPage = (rcMatches[2] - rcMatches[1]) + 1;
   var totalProducts = pcMatches[1];
   var totalPages = Math.ceil(totalProducts / productsPerPage);
   var meta = {
      categoryID: categoryID,
      productsPerPage: productsPerPage,
      totalProducts: totalProducts,
      totalPages: totalPages,
      isModern: false, //special key for category pages with modern faceting and filtering,
      pages: []
   };
```

```
        // the new walmart category pages have a special 'Refine by' drop down missing in
        // old ones. So detect that and process page accordingly.

        if($(".dropHeader").text().indexOf("Refine by") !== -1) {
            meta.isModern = true;
            var redirURL = url.parse(response.request.uri.href);
            redirURL = redirURL.protocol + "//" + redirURL.host + redirURL.pathname;
            for (var i = 0; i < totalPages; i += 1) {
                var pageNumber = i * productsPerPage;
                var pageURL = redirURL + "?ic=" + productsPerPage +  "_" + pageNumber;
                meta.pages.push(pageURL);
            }
            deferred.resolve(meta);

        } else {
            for (var i = 0; i < totalPages; i += 1) {
                var pageURL = "http://www.walmart.com/cp/" + categoryID +
"?showAll=true&bti=" + i;
                meta.pages.push(pageURL);
            }
            deferred.resolve(meta);
        }
```

```
            print.info(meta);
        } else {
            print.error(categoryID + " - Error getting page count from: ->" + recordCount + "<-");
            deferred.reject(new Error(categoryID + " - Error getting Page Count"));
        }
    }

});
return deferred.promise;
}
```

APPENDIX C – CATEGORY PAGE SCRAPPER

This code fragment is from the middleware data transformer. It traverses through each category page and extracts all the items on the page. Each item is then added onto the product queue.

```
var getProductsInPage = function (categoryPage) {
    var deferred = Q.defer();
    var categoryID = categoryPage.category.id + "_" + categoryPage.pageNumber;
    var categoryURL = categoryPage.url;
    print.info(categoryID + " - Requesting: " + categoryURL);

    request(categoryURL, function(error, response, body) {

        if (error || response.statusCode !== 200) {
            print.error(categoryID + " - Request Error : " + error);
            deferred.reject(new Error(categoryID + " - Request Error"));
        } else {
            print.info(categoryID + " - Response From: " + response.request.uri.href);
            var $ = cheerio.load(body);
            var products = $(".ListItemLink");

            if (products.length < 1) {
```

```
            deferred.reject(new Error(categoryID + " - No Products found on page"));
        } else {
            var meta = {
                categoryID: categoryID,
                products: [],
            };
            products.each(function(i, elem) {
                var productName = $(this).text();
                var productID = $(this).attr("href").split("/").pop();
                meta.products.push({
                    name: productName,
                    id: productID
                });
            });
            // print.info(meta);
            print.info(categoryID + " - Found " + meta.products.length + " products ");
            deferred.resolve(meta);
        }
    }
  });
  return deferred.promise;
}
```

APPENDIX D – PRODUCT SCRAPPER

This code fragment is from the middleware data transformer. It traverses through each product page and extracts all available item metadata. This structured information is then sent to a search indexer where an inverse document frequency table is built.

```
var getProductMeta = function (product) {
    var deferred = Q.defer();
    var productURL = "http://www.walmart.com/ip/" + product.id;

    print.info(product.id + " - Requesting: " + productURL);

    request(productURL, function(error, response, body) {
        if (error || response.statusCode !== 200) {
            print.error(product.id + " - Request Error : " + error);
            deferred.reject(new Error(product.id + " - Request Error"));
        } else {
            print.info(product.id + " - Response From: " + response.request.uri.href);
            var $ = cheerio.load(body);
            var productName = $("h1.productTitle").text();

            var price = $(".bigPriceText1").first().text() + $(".smallPriceText1 ").first().text();
            if (price) {
```

```
    price = price.replace("$","").toNumber();
  } else {
    price = null;
  }

  var rating = $("#BVRRSourceID [itemprop=ratingValue]").html();
  if (rating) {
    rating = rating.toNumber();
  }

  var reviews = $("#BVRRSourceID [itemprop=reviewCount]").html();
  if (reviews) {
    reviews = reviews.toNumber();
  }

  var image = $("#mainImage").attr("src");
  var description = $("#prodInfoSpaceBottom").text().replace(/(\r\n|\n|\r)/gm,"
").replace(/\s+/g," ").trim();

  var relatedItems = $("#irs_bottom a");

  var specs = {};
  $(".SpecTable tr").each(function(i, elem) {
```

```
            var tds = $(this).find('td');
            var key = tds.first().text().trim().replace(":", "");
            var value = tds.last().text().trim();
            specs[key] = value;
        });

        var meta = {
            name: productName,
            id: product.id,
            price: price,
            reviews: reviews,
            rating: rating,
            image: image,
            specs: specs,
            description: description
        }

        print.info(meta);
        deferred.resolve(meta);
    }
  });
  return deferred.promise;
}
```

APPENDIX E – ADDITIONAL ONTOLOGY DIAGRAMS

Following are additional supplementary diagrams of the Wal-Mart and Target ontologies. Unlike the ontology diagrams illustrated earlier, the nodes in these graphs have been labeled. Some close ups shots are also included.

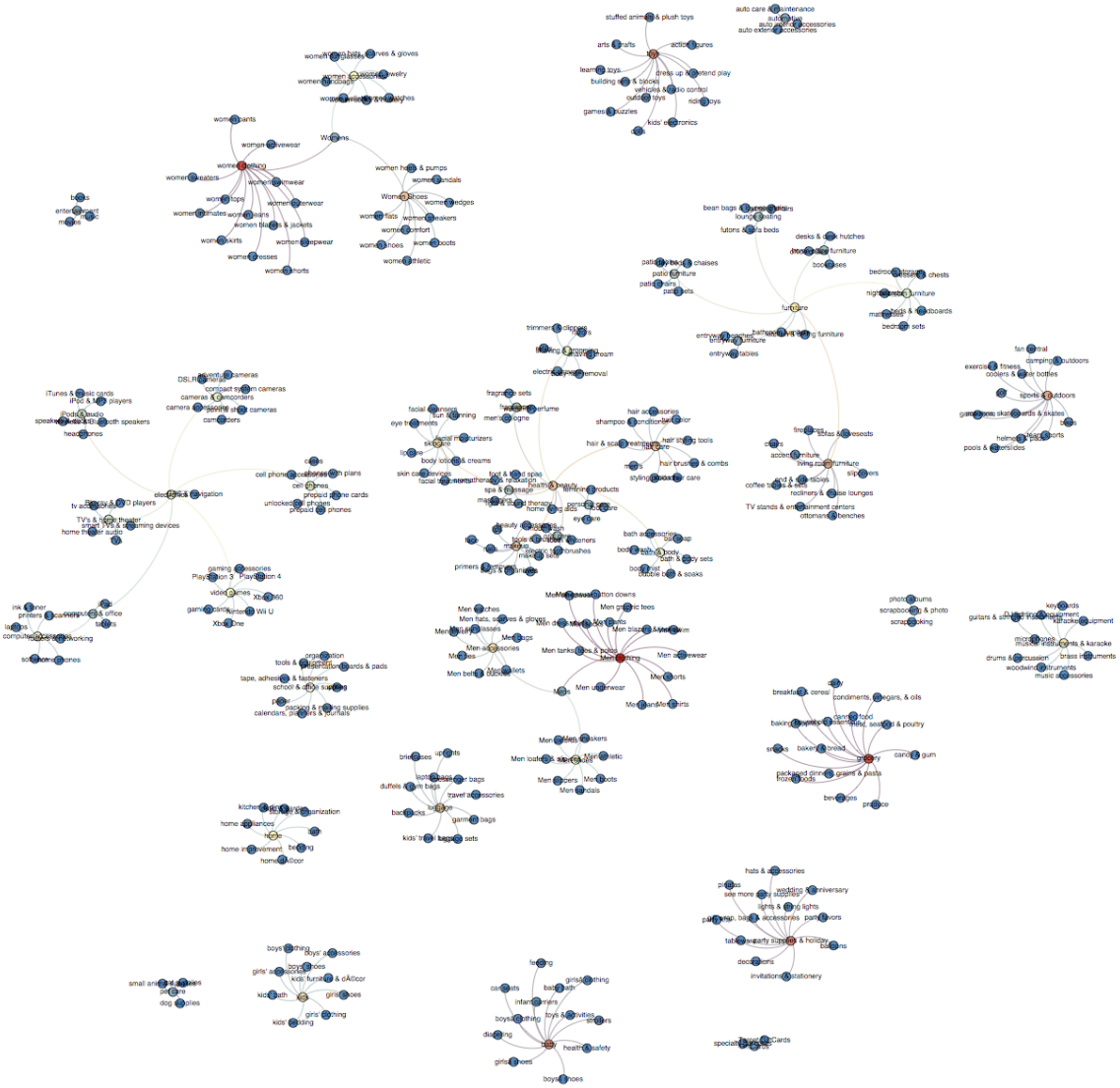

Figure 26: Labeled Target Ontology

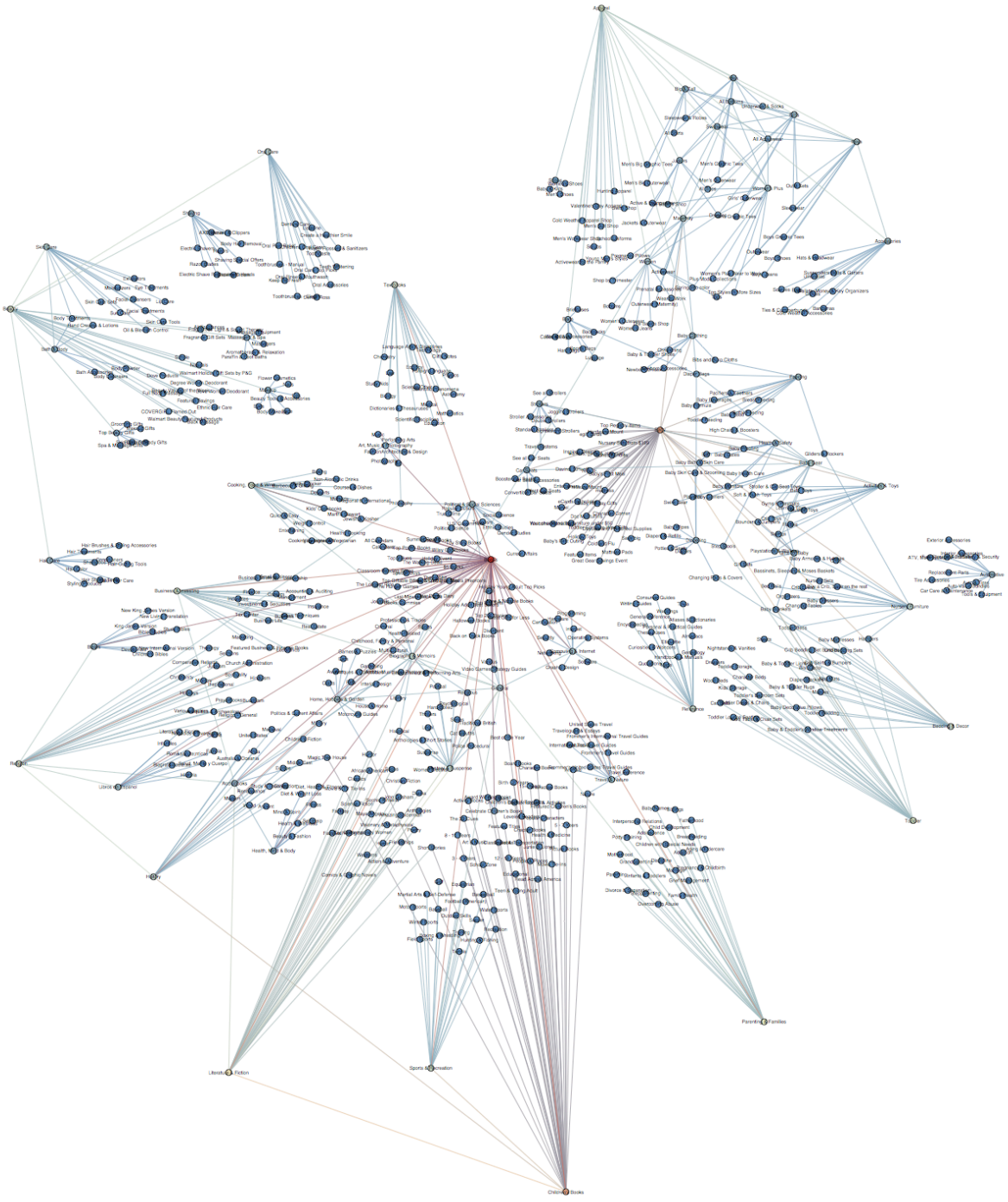

Figure 27: Labeled Wal-Mart Ontology

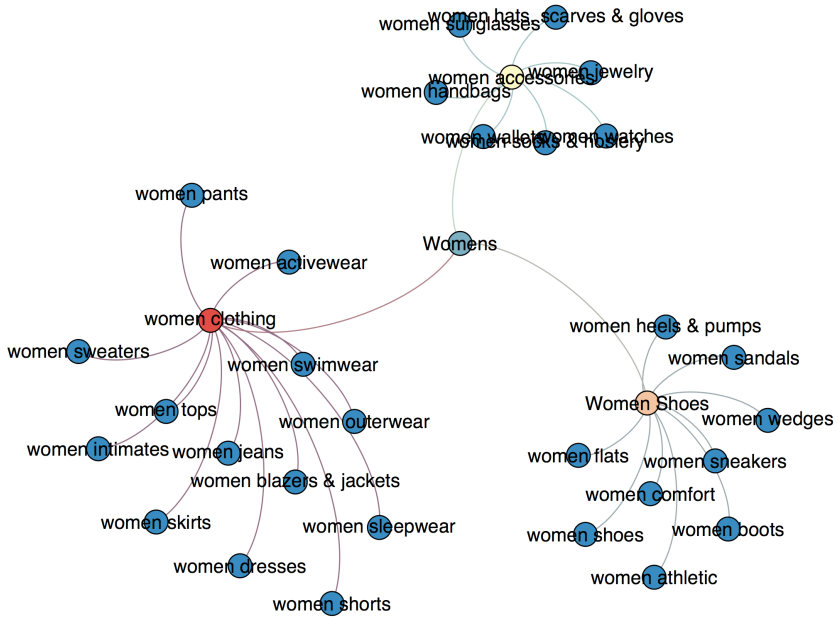

Figure 28: Target - Women's Apparel

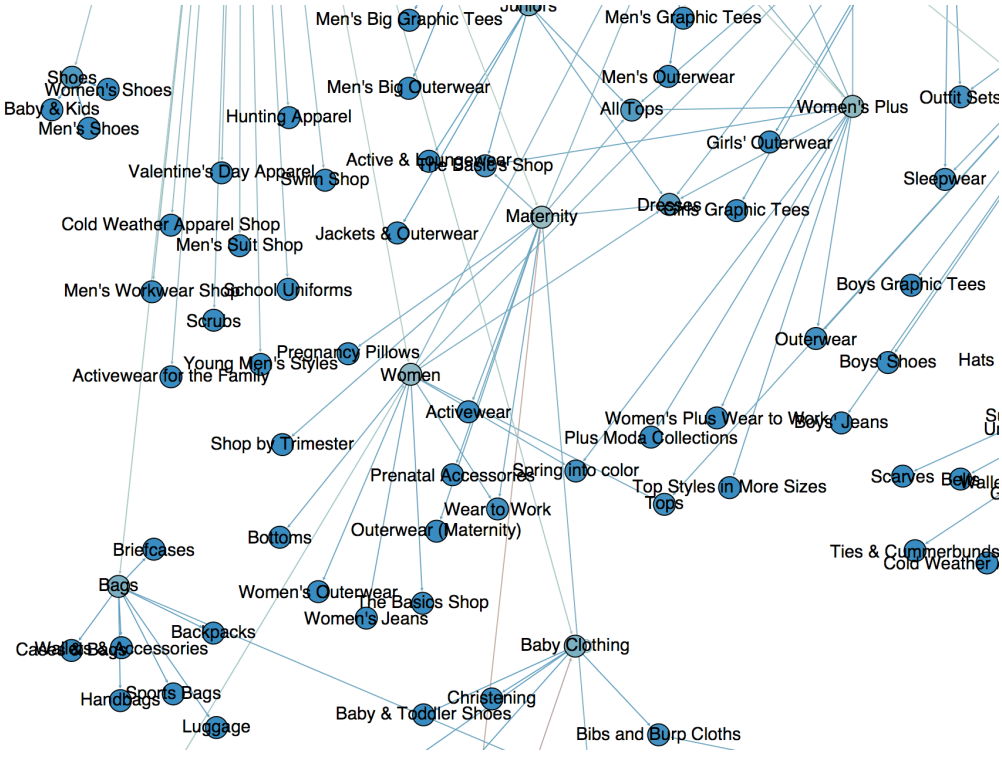

Figure 29: Wal-Mart - Women's Apparel