Seven Years of Wikipedia's Revision History as a Time-Dependent Graph: A Love Story

Ryan Noon, Stanford University CS345A

Winter Quarter 2010

Introduction

Last quarter I took CS322 (Professor Leskovec's network analysis class) and I learned that many very interesting things in the real world can be modeled as graphs. For my final project in that class and friend and I logged and modeled scientific conferences on Twitter (using the pre-shared Twitter *#hashtags*). We implemented a "time-dependent graph" data structure, which was essentially a collection of updates and a policy for adding/removing nodes that allowed the user to render the state of the scientific conference at any time. The data structure supported interactive rewinding and fast-forwarding, allowing for traditional graph metrics to be easily performed over the lifespan of the data.

This quarter in CS345A I decided to declare war on huge datasets. I was curious to see if my data structure from last quarter could be made to efficiently scale to the size of massive graph-able datasets. The dataset I chose was the entire 2.7 terabyte edit history of Wikipedia. This dataset includes the full text of every revision ever made to any article, talk page, user page, or template ever hosted by Wikipedia in the English language over the period 2001 to early 2008. It contains 11,405,052 articles and the 167,464,014 revisions of the complete-text record of these articles. It is truly a massive dataset.

1. Wikipedia as a Gigantic Graph

1.1 Mapping to a Directed Graph

My model for Wikipedia as a time-dependent directed graph is as follows: all English language encyclopedia articles present in Wikipedia at a given time are allowed to be nodes in the graph. Utility/meta pages (such as talk pages, disambiguation pages, category pages, user pages, and templates) are excluded. An article joins the graph the moment a revision is made to it and it can never leave (garbage collection based on the absence of any inlinks was considered but ultimately abandoned for performance reasons). An edge from node A to node B represents an intra-Wiki outlink from article A to article B. These edges join the graph when a revision is made to the article that adds this outlink, and they would leave should a later revision remove the outlink.

1.2 Motivations

My motivation for spending countless hours build such a dataset and data structure comes both from my past experiences with the encyclopedia and my technical ambition as a programmer. I have always felt close to the Wikipedia project and have been reading, editing, and contemplating humanity's greatest single knowledge aggregate since 2003. I have spent entire days reading Wikipedia: moving from article to article, traversing the massive strongly connected component at its center and expanding my awareness of the grandiose interconnectedness of human history and thought. As a senior in high school in 2004 I even wrote my Stanford undergraduate application essay about the broad possibilities and subtle challenges of the pluralism and wide accessibility of knowledge that Wikipedia embodies. This project gave me a chance to create a dataset for me and others to use to quantitatively explore the formation and explosive growth of Wikipedia.

The benefits of this type of analysis are three-fold:

- Studying the structure of the way humans categorize information on Wikipedia allows us to concretely quantify the interconnectedness of concepts that would otherwise be hard to describe. This information could be used as training data for machine-learning systems to better understand the world.
- Having the ability to rewind and fast-forward Wikipedia gives the analyst a high-level summary of human affairs that is updated nearly-instantaneously to reflect the current state of the world. The ramifications of specific events can be analyzed in a controlled-environment. For example, the Wikipedia edit history contains the complete rise of Barack Obama as a national figure. From his first entry on March 18th 2004 as an Illinois State Senator to his election as President of the United States at the end of 2008, we can analyze the popularity and connectedness of his local subgraph over time to better understand the crucial events and associations that precipitated his role in recent U.S. history.
- The dataset encompasses the initial conception and formation of Wikipedia which is itself a communication and organizational task unprecedented in human history. How did an initially small (but ever-growing) group of human beings decide to communally construct the world's greatest encyclopedia? How did the thousands of 21st century citizens who have since contributed decide to prioritize and distribute the task of describing the most important parts of all of the things that our brains know how to write about?

It should be clear that Wikipedia is immensely valuable not just for its content but for its structure. The topology of interconnected discrete kernels of knowledge represents a deeper, more communal form of knowledge. My goal in this project was to raise more questions than I could possibly answer and to build a robust dataset with an intuitive set of tools for myself and others to use in the future.

2. Implementation

2.1 The Raw Data

As of the time of this writing the largest database dump that Wikipedia has ever offered is the enwiki pages-meta-history archive, which was last compiled at the beginning of 2008. The reason this dataset is compiled so seldom must be because of its immense size: uncompressed the dump yields approximately 2.7 terabytes of XML. This XML follows an available schema and is essentially a list of articles where each article encompasses at least one revision. Revisions are labeled with a unique id number, the revision author, a UTC timestamp, and the full text of the revision in Wikimedia's markup syntax. Additionally, a revision can be marked as **`<minor />`**, indicating that it was generated automatically by a crawler or other procedural editor.

Each of these full-text revisions comprises the raw material necessary to construct a time-dependent graph, but each still requires a decent amount of processing to get to that point. The time-dependent graph system I developed last quarter works based on explicit atomic diffential-updates that can be applied in forwards and reverse. For a Wikipedia-oriented time-dependent graph our ideal update would look like this:

```
<article_id> <revision_id> <timestamp> <outlinks_gained> <outlinks_lost>
```

In order to obtain this "holy grail" representation I decided to make two passes over the data. The first pass would be "hard and fast", filtering the raw outlinks from every article and figuring out which ones were new and which were old. These outlinks are very precarious, however, as Wikipedia internally resolves outlinks via a search-based mapping and not a publically accessible identifier. The second pass would operate on the (much reduced in size) intermediate data to assign absolute identifiers to every article and resolve as many outlinks as possible to these identifiers.

2.2 The First Pass: High Speed, Low State

Although I have Amazon EC2/EBS and my own multi-terabyte array at my disposal, my tests showed that the CPU requirements to decompress the XML on the fly were small enough to yield superior IO performance from streaming directly to my parser than from reading the already uncompressed file. In terms of the parsing algorithm, I first considered extensive parallelizing via Map Reduce, and indeed my first attempts at this task simulated Map Reduce via parallelized multiprocessing. Eventually I realized, however, that the requirements of the dataset limited parallelizability: the necessity of maintaining per-article revision flow meant that the workload could not be easily split into smaller chunks than entire article histories. For example, a round-robin system of splitting individual revisions to different sub-parsers would guarantee good utilization (because most revisions are within an order of magnitude of each other in size) but it would take at least one additional step to reconcile the history of the outlinks for each article.

The simplest easily-parallelizable solution would thus have been to have each sub-parser parse the history of a given article. Wikipedia's particular distribution of article history size makes this approach completely pathological to performance. On Wikipedia, popular articles are edited far more often than unpopular ones: the distribution of article revision histories appears to be governed by principles reminiscent of a Barabási–Albert process where the "rich get richer". The end result seems to be an extreme power-law distribution. This makes the data we're passing through our queue to sub-parsers extremely heterogeneous in size, making it extremely likely that other sub-parsers will finish their (relatively cheap) parsing jobs and starve waiting for a large article to pass through the input bottleneck.

The approach I eventually settled on minimized IO overhead by using one extremely fast thread with constant state to parse the XML. I tossed out the Expat XML library in favor of my own state machine and I carefully mixed lean regular expressions with fast string operations and loop lookup optimizations to build what may be the fastest Python I have ever written. The end result was an average parser throughput of 38 MB/sec on my Core i7 920 workstation. For reference, this is about three times slower than the day-to-day speed of my local RAID-0 disk array. In this factor of three slowdown my parser was able to automatically disregarded irrelevant article types, strip article outlinks, and diff them against the previous revision for the current article. Because of the built-in state machine and fixed data structures, I measured memory usage at no more than 6-7 MB (including interpreter) during this operation. At this speed my pass through the massive Wikipedia XML dump took only 20 hours to complete. The end result was two files: a list of all of the article names and a condensed list of revisions as described above (albeit with nebulous strings instead of absolute identifiers). After several days of different designs and parallelization schemes, it was completely unexpected that a tight, fast single threaded solution could perform so well on this specific dataset. In this step, the size of the data was chopped from 2.7 terabytes to a mere 15 gigabytes.

An interesting hack I had to employ was using #'s to delimit the lists of outlinks in my intermediate files. Wikipedia article titles are not limited in the characters they can use in their titles, but due to URL formatting the "#" is guaranteed to be reserved for anchor tag lookups. There really are Wikipedia articles with every other delimiting character I tried.

2.3 Cleaning, Identifying, and Resolving

The next step in the process initially seemed straightforward but ended up being somewhat taxing. If I wanted to have any hope of compact representation or efficient querying, I needed to assign every article a unique id number and resolve every outlink to one of these numbers. The first step was to go through my file of article names and assign each a number. During this process I tossed out any undesirable article types (like talk pages or auto-generated lists) that had made it through my first round of filtering.

Next came the arduous task of resolving outlinks. This is especially difficult because Wikipedia is designed for easy editing where users can enclose any topic in square brackets and Wikipedia will automatically resolve it upon click to an article using their own internal search. Thus it was necessary to form a heuristic to resolve as many of these outlinks as possible. I loaded all of the article names and new identifiers into a hash map and began experimenting for several hours with different heuristics. I eventually settled on the following process:

1. Try the name itself, stripped of any extra whitespace on the side.
2. If unsuccessful, try capitalizing every word in the name (where a word is defined as characters separated by spaces).
3. If unsuccessful, try capitalizing every word at a hyphen level.
4. If unsuccessful, try replacing every hyphen with a space.
5. If still unsuccessful, try the title in all lower case. I wanted to save this for last because Wikipedia article titles are case-sensitive and disregarding all case information would cause a lot of false positives.
6. If unsuccessful, give up and reject the outlink.

Additionally, when capitalizing every word in a title, common words such as "of", "the", "a", and "in" were left uncapitalized so as not to miss important articles like "Government of Austria". After every revision I would pour through the rejected sample and look for cases that I really should not have rejected. After about a dozen iterations I was satisfied: my "reject file" contained some of the most egregious spelling mistakes and hilarious personal insults that humanity has ever seen. I also maintained the policy of rejecting any revision that had all of its outlink changes rejected. With my heuristic I was able to resolve and cleanse all 69,850,248 Wikipedia revisions in about 91.5 minutes at a rate of 2.84 MB/sec. Of these revisions, 75.47 percent of them were accepted, and of the 477,657,442 outlink changes that these revisions contained my heuristic was able to resolve 67.84 percent of them. These numbers initially seemed low to me until I realized exactly how much filtering needed to happened, including outlinks to versions of Wikipedia in other languages.

The end result of this step was a list of revisions where every string had been replaced by a hexadecimal identifier and all outlinks were guaranteed to be valid. The entire dataset now contained approximate 3.8 gigabytes of uncompressed text.

2.4 Designing an Efficient Archive Schema

The next task was to design a database schema that would allow me to quickly query into these revisions in the order in which they occurred (which was different than the order they were presented in the original XML). For a DBMS I decided on SQLite because its read performance in asynchronous mode was on par with MySQL and it would allow my data great portability and compressibility.

After a few iterations I realized that performing a sort of these revisions without Map Reduce would be practically impossible. I worked around this by coming up with the following schema:

- **`TABLE timestamp_to_rev (timestamp INT PRIMARY KEY ASC, revids TEXT)`**
- **`TABLE revs (revid INT PRIMARY KEY, time INT, article INT, outlinks_gained TEXT, outlinks_lost TEXT)`**
- **`TABLE article_names (article INT PRIMARY KEY, name TEXT UNIQUE)`**

There are a lot of design wins here. Given a specific timestamp (which I held in chronological order) I could (in constant time) have a list of all of the revisions (stored as a delimited string) that occurred at that timestamp. For each one of these revisions I could (also in constant time) have the article that was revised as well as the article ids of the outlinks that had changed. Loading the timestamp to revision table was somewhat challenging as it required me to have a hashmap with many millions of entries.

2.5 Manipulating Time

My ultimate goal was the construction of an interactive shell where the user could quickly load up my dataset and reload the state of Wikipedia at the last place she left it. To accomplish this I created a WikiGraph time-dependent graph object that handled all database access and supported methods like **`get_available_time_range()`**, **`get_current_graph()`** and **`render_to_time()`**. The method **`get_current_graph()`** returns the instantaneous state of the WikiGraph, which is stored internally as static graph object (more on this later). This is similar to the design and functionality I built into my Twitter analytics library from last quarter, just scaled up tremendously for a dataset orders of magnitude larger.

Rendering the graph entailed taking the current state of the graph and figuring out which timestamps needed to be visited in which order (for moving forwards and backwards). To aid this, execution of the shell triggers a 15 second caching of all of the available timestamps in the database into a densely packed array object. Since the result array is in sorted order, figuring out the sub-array to be traversed can be accomplished with O(log N) bisection algorithms and a little tricky logic. The actual rendering algorithm is fairly simple: for each timestamp in our steps ask the database for the revisions that happened at that second. For each of these revisions, ask the database for the revision record. For each revision, if the article being revised has never been seen before add a node for it to the graph. For each outlink added or removed notify the graph of the corresponding gain or loss of the edge. If this node claims an outlink to an edge that has not yet been seen, disregard the edge. Lastly, if we're travelling backwards in time reverse the list of timestamps and reverse each list of revisions to ensure the correct application order. Also, if we're travelling backwards in time reverse the meaning of added and removed outlinks. Following this algorithm in a highly optimized loop with aggressively cached databases makes it possible to move quickly through large, densely packed stretches of time.

2.6 Designing the BigGraph

The last crucial piece necessary for this data structure was the internal graph data structure that could support rapid additions and subtractions of tens of millions of new nodes and edges. This part, like all of the others, found me iterating through multiple designs before eventually setting on an aggressively cached database affectionately named the BigGraph.

My first attempt found me using a NetworkX graph to store the instantaneous state. These graphs are excellent at efficient operations and support all sorts of built-in analysis. The downside is in memory bloat. Internally they store the graph as a hashmap of nested hashmaps. The memory overhead involved in this was fearsome, with main memory quickly being overloaded with edges only a year or two into the edit history. My next idea was a MiserGraph, or essentially the most compact representation of a graph keyed on numerical node ids that could possibly be constructed in Python. It featured one central hashmap with a hashset for each node. This afforded me graphs of up to several million nodes and edges that still supported constant time modifications. However the contiguous memory footprint of even this structure quickly overwhelmed my workstation. Had I used an Amazon EC2 machine with many more gigabytes of RAM I could have stopped here, but I had the stated goal of allowing effective analysis to take place on ordinary computers.

My crucial insight came when I again remembered Professor Leskovec's lectures from last quarter on real-world graph behavior. Most real-world graphs are not accessed randomly: preferential attachment in the Barabási–Albert mode dominates many important networks. This insight, combined with my systems background, led me to believe that a graph data structure with a comparatively small cache could perform very well on datasets that never had a chance to fit in memory if their access/construction/deconstruction patterns followed real-world behavior. This led me to the BigGraph, where a central hashmap associative cache stores hashsets (like the MiserGraph). The cache, however, detects when the total number of edges in the cache is over a specified limit and randomly selects a node to eject in constant time. Ejection causes the node and the hashset (serialized via the extremely quick cPickle module) to be written out as the key and blob fields of a special archive database table. Limiting the cache based on edges instead of nodes (even though the structure caches complete nodes) was a decision I made based on my observation that the most important nodes would likely have the most edges, making any sort of uniform or predictable memory usage impossible. With this data structure in hand, my WikiGraph was finally ready for testing.

3. Experimental Results

3.1 Software performance

Only a few initial time-based metrics have been performed with the new dataset, but performance of the tools themselves is highly encouraging. Rendering the state of Wikipedia from the very beginning to the very end is (naturally) IO limited and takes merely 4.5 hours on a single (non-RAID) consumer-grade hard drive. Over the course of a single 4.5 hour run, the WikiGraph computes the effect on graph structure of 35,035,869 individual revisions at an average performance of 2157.67 revisions per second. For many hops over a small region, SQLite's integrated caching allows for performance an order of magnitude higher. Solid-state drives that allow for orders of magnitude more IO operations per second could potentially even make this traversal CPU-limited. To get a sense of the sheer number of timesteps involved in this operation, of the 219,888,449 "real-world" seconds that occurred from the first revision to the last in this dataset, 29,135,968 (or 13.25%) of these seconds saw at least one meaningful human edit to Wikipedia.

Currently the code supports integrated reporting of basic graph metrics (such as the number of nodes, edges, the cache churn, and the cache hit/miss statistics). This framework is extensible as user-supplied callbacks are also supported. This means that the user can supply a function and an interval at which to run it and the WikiGraph framework will automatically run the function on the graph and publish the results to a file. This makes it simple to create batch processes that walk back and forth through the dataset performing any arbitrary analysis on the graph.

3.2 Preliminary Numbers

The following analysis was conducted on a single pass through the dataset using the WikiGraph Python shell to specify a default report and to render the graph at the last available timestamp. Statistics were sampled approximately every two days during the walk. The cache-size was set to seven million edges, which translated to a final maximum memory footprint of 532 megabytes. At this size, the final cache hit percentage came out to 77.8 percent on 36,079,348 total accesses after the cache filled in June of 2005 (around half-way through the render). The machine used to render the graph was a Core i7 920 at 2.66 gigahertz with 6 GB RAM and the disk that held all data was a single two-platter 640 gigabyte consumer-grade Western Digital 7200 RPM Caviar Black.

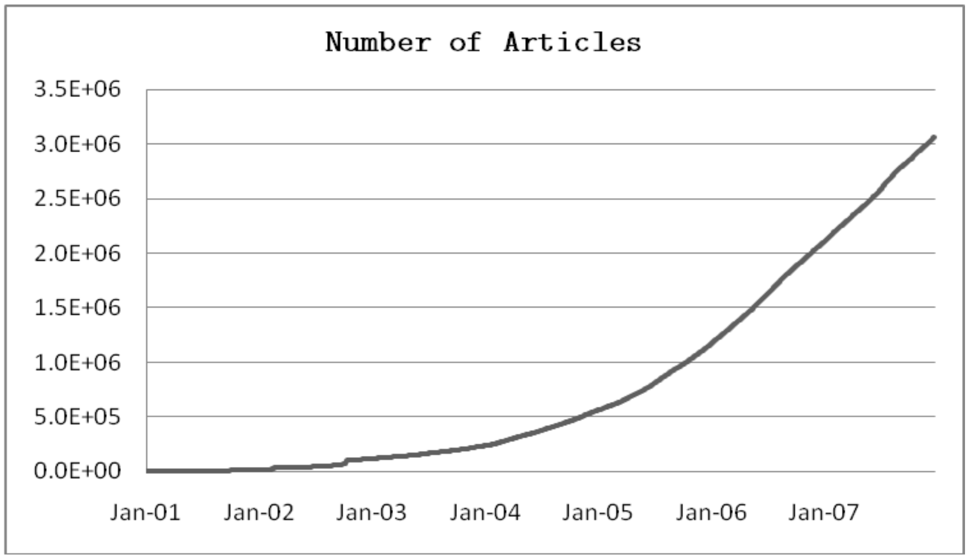

Figure 1: It is clear that Wikipedia really exploded sometime in 2005 and by 2008 showed no sign of stopping.

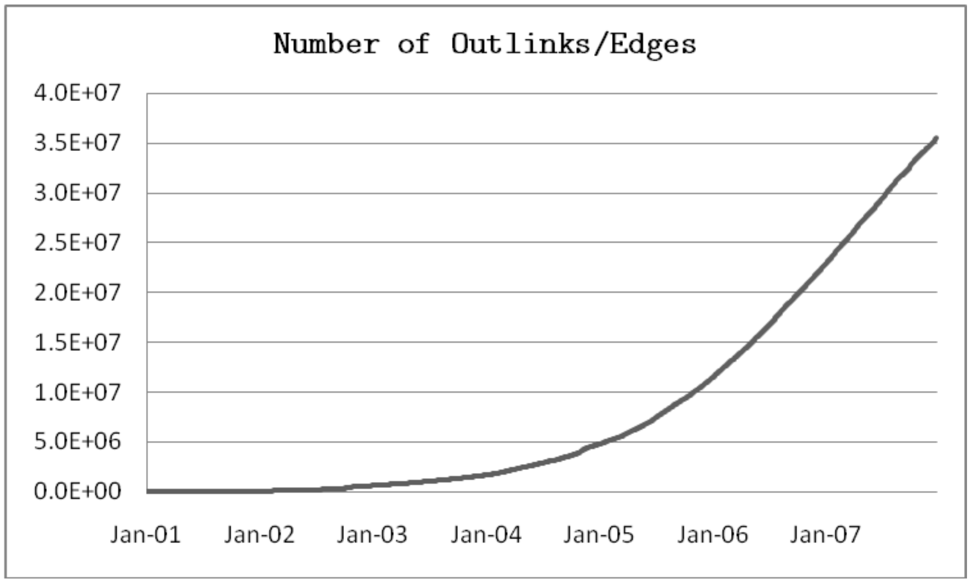

Figure 2: Wikipedia is a fairly sparse graph, but it still had 40 million inter-article outlinks by 2008.

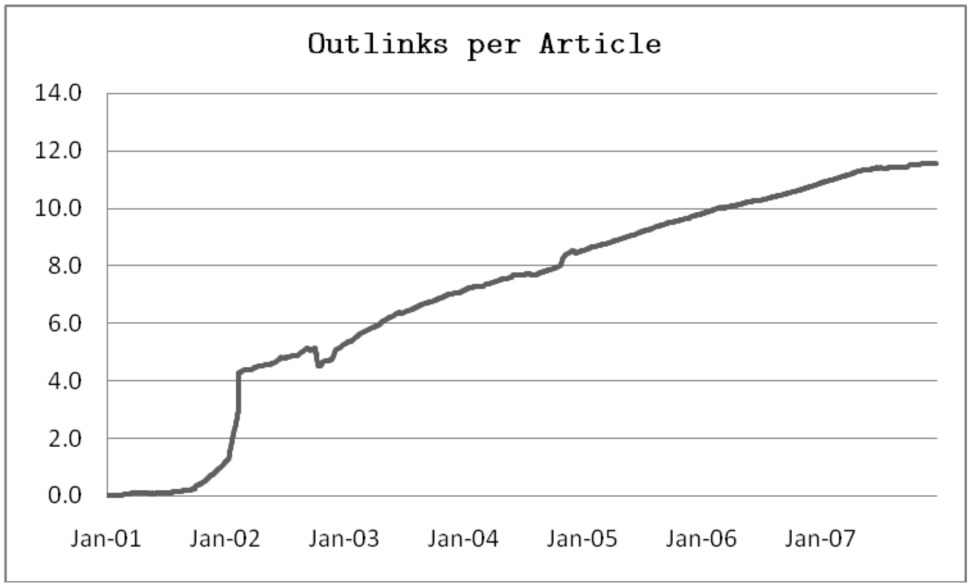

Figure 3: Although Wikipedia is fairly sparse it is getting denser all the time. This metric indicates that the average Wikipedia article is becoming more complex over time. The addition of new articles makes the average

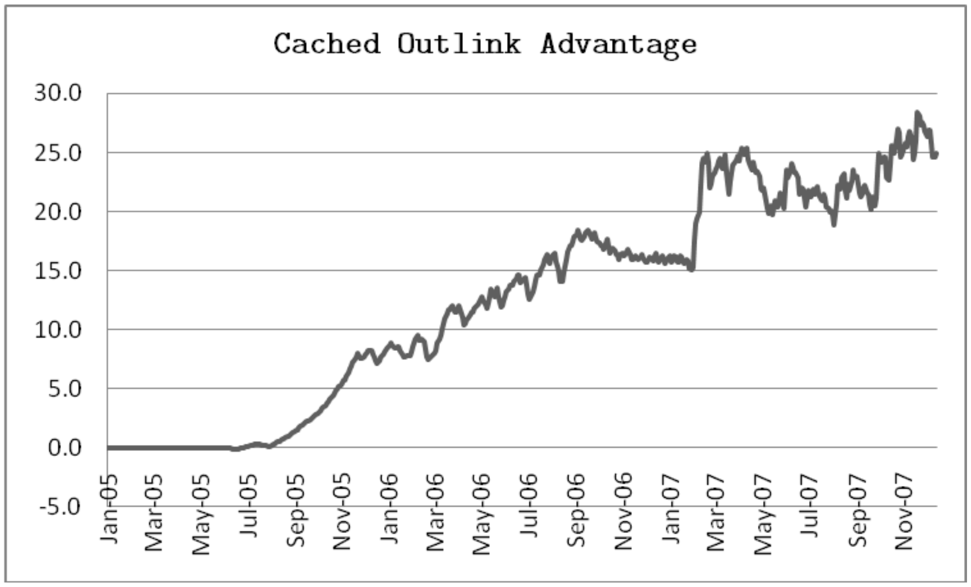

Figure 4: This plot shows the difference in edges-per-node for nodes in the cache versus other nodes. The cache does not fill until June of 2005, at which point the nodes that remain in the cache gain a significant and ever-increasing advantage over other nodes. This indicates that the most in-demand nodes on Wikipedia are those with the highest number of outlinks, reinforcing the observation that on Wikipedia, the richest articles consistently get richer.

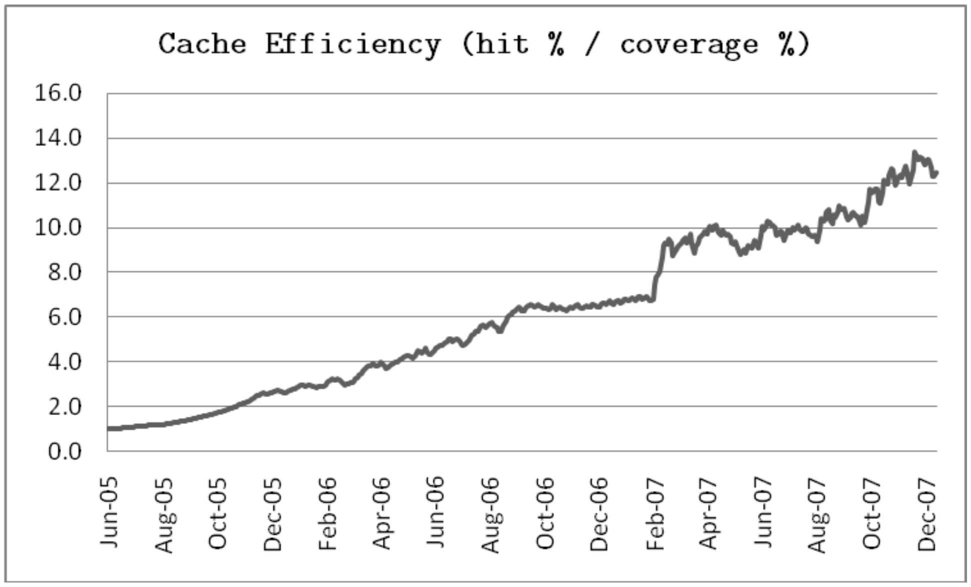

Figure 5: Here we can see the efficiency of the cache (defined to be the hit percentage over the coverage percentage) from when the cache first becomes engaged in June 2005 (when the cache hits 7 million edges). As the number of articles continues to grow our coverage naturally drops but we can still maintain an average hit percentage of 77 percent on new additions to the graph (and the trend has this hit percentage essentially leveling off). This indicates that the vast majority of new articles being added to Wikipedia are not frequently revised. The final cache coverage is 6.2% and is steadily dropping.

3.3 The BigGraph cache as an automatic summarizer

An interesting side-effect of this implementation that deserves further research was the tendency of temporal locality in caching to automatically identify the most important parts of Wikipedia. After the first full Wikigraph render, I analyzed the 118,000 nodes in the BigGraph cache. I found a strongly connected component of 95,000 of them and a quick informal analysis of consistently popular articles identified all of them as being present in it. I would like to run more analysis on this tendency to see exactly how good of a probabilistic summary it is and to see if it can be improved with other cache ejection policies.

4. Future Work and Conclusion

The analysis conducted so far has only scratched the surface of what is possible with this dataset and tools. The engineering challenges of cleansing very noisy data and the coding battle that was designing scalable high-performance data structures consumed a tremendous amount of my finite time. I could spend an entire quarter playing with this data (and I plan to). In the near future I would like to conduct traditional link and cluster based analysis over the graph structure to build up a sense of article importance over time. I would also like to tie-in support for automatic graph partitioning with Hadoop to allow for distributed application of those graph algorithms that are easily parallelizable.