**OutOfMemoryError: Final Report**

**Vamsee Krishna, Leo Thumma, Nirtyanath Jaganathan, Samuel Stern**

**CIS 455/555 Spring 2012**

**Introduction**:

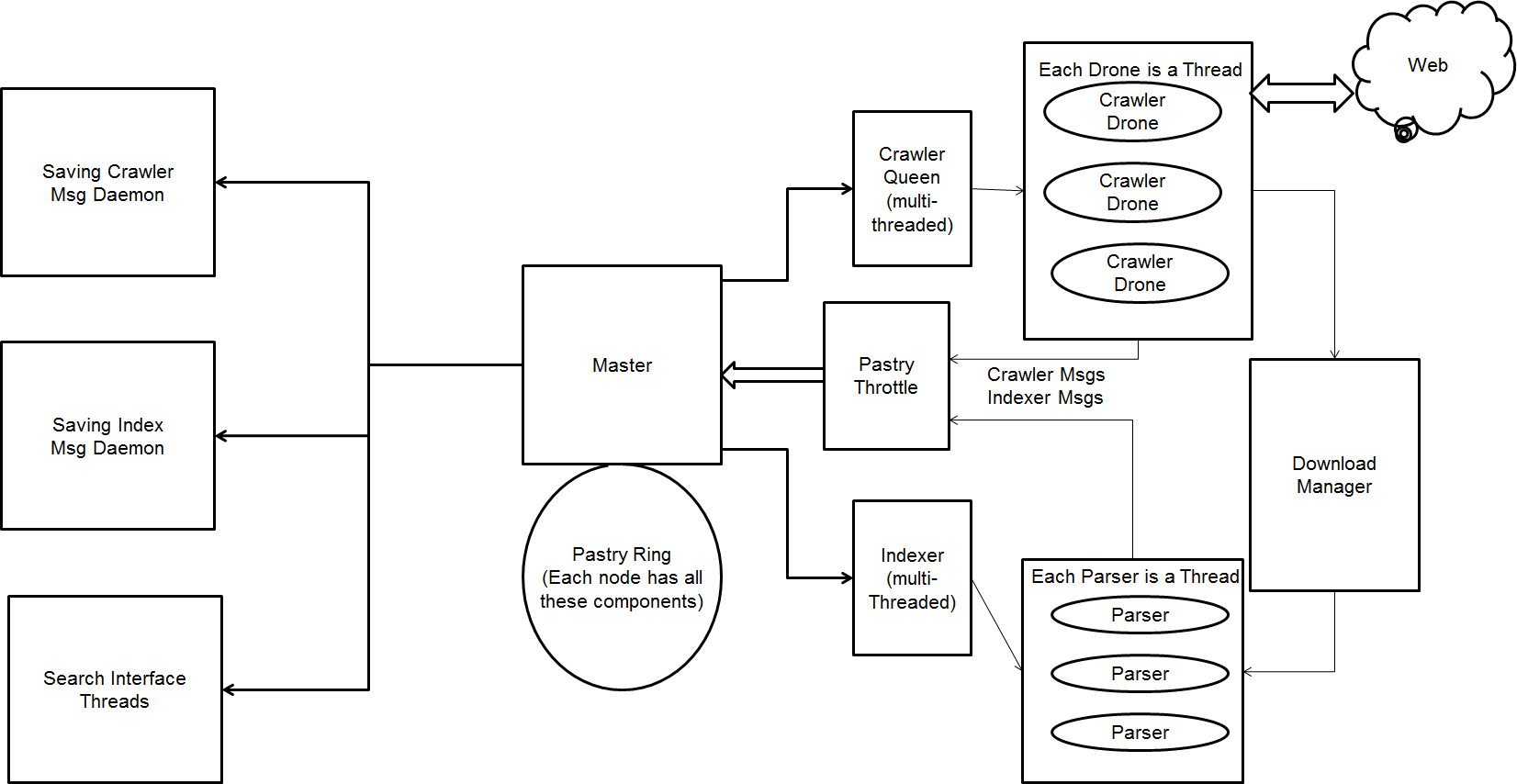
The goal of this project was to implement a complete search engine that mimics the architecture of the Google search engine outlined in the paper *The Anatomy of a Large-Scale Hypertextual Search Engine* (Brin/Page, 1998).  However, the search engine that we have designed and implemented makes use of modern technologies and frameworks that were not available to the original developers of Google in order to improve performance and ease of implementation.  Our approach to achieving the project goal can be summarized in four basic tasks: crawl a large subset of the web, index that subset by content, provide an absolute ranking for each page encountered, and create an interface that uses the compiled data to quickly provide relevant responses to user queries.  All four of these tasks were completed successfully on a page corpus containing 350,000 to 500,000 unique pages by the implementation outlined in this report.

In the original plan for the project, we outlined a strict division of labor and concrete milestones.   The division of labor- (Sam-Crawler, Nirtyanath-PageRank, Vamsee and Leo-Indexer, Search) lasted only until the first attempt at integration, during which we began to understand the advantages of group development.   After the basic crawler, indexer, and search were completed separately all development was done as a group, and concrete milestones largely disappeared in favor of a more agile development style where features were integrated immediately upon completion.  The result, outlined in the rest of this report, is a highly robust product in which all features work together seamlessly and no component hampers performance of any other.

**Project Architecture**:

**General Structure**:

The program was based on running ten identical nodes; each loaded with the same Java program and backed by its own database. The nodes coordinated over the network in order to organize action but no replicated information exists between nodes. The nodes were largely coordinated using a key-based routing framework, with information retrieval requests moving over the same network.



**Figure 1-System Architecture**

**Crawler**:

The crawler ran an identical program on each node of the system.   The program contained a single coordinating “queen” with a variable number of “drone” crawlers that could each crawl a single page simultaneously.   Coordination between crawler nodes was done using pastry’s key based routing which assigned a consistent partition of the URL namespace to each node.   Each crawler node was backed by a transactional database containing the URL frontier and information about pages already crawled.   Once crawled, pages were handed off to the Indexer for parsing while a list of URLs contained in the page was delivered to the PageRank system for organization.

**Indexer**:

The indexer also ran an identical, multithreaded program on each node of the system. The indexer works in parallel with the crawler by taking each crawled page and extracting each word from the page. Each word is given a weight according to the structure of the page and stored along with the page URL where it was found. The indexer relies on key-based routing for organization and a database for persistence. The indexer stores the useful page information to disk and discards the rest, rather than needlessly saving the entire content of every page encountered.

**PageRank**:

For each item crawled, the PageRank accepts the URL along with a list of the URLs linked from that page. Using the PageRank algorithm on MapReduce platform, the PageRank assigns a ranking to a URL as well as a ranking to all links in the list. Once computed, the values are distributed to all nodes in the system for use in searching.

**Search**:

The search for a query is done in several layers with extensive multithreading to ensure that no part of the search can bottleneck the process. The user’s query triggers creation of threads that fetch out the indexed data and rank it using TF/IDF, context weight factor and proximity using indexes and fuse these results together into a final score by some predetermined ratio. The resulting score for each link is then combined with the PageRank score for the link in order to form a final score.  These final results are sorted and interleaved with other outside search APIs before being returned to the user.

**Implementation**:

**Crawler**:

The crawler architecture was a modified version of the Mercator architecture, pastry was used to organize it.   Each node contains a single CrawlerQueen and a varying number of CrawlerDrones.  The drones constitute a thread pool and accept crawling assignments one page at a time from the Queen.   Upon receiving a URL the drone first checks for Robots.txt disallow and/or crawl delay restrictions, then sends a HEAD request to ensure the validity of the page, and the finally crawls the complete page. The HEAD request checks that the page content type is HTML or plaintext, the language is declared as “en’ (or some variant of en) and that the response code of the request is in the 2XX range.  The content of each page is sent as a String to the Indexer, while the links contained in the page are normalized and then forwarded both to the local queen and to the PageRank.  Page downloading and URL normalization are both performed by functions included in Java libraries, specifically making use of the HttpURLConnection class for fetching and the overloaded URL constructors for normalization.  The drones contain minimal logic, completely trusting the queen to make proper crawling assignments and to handle the information extracted from each page.  When a page is successfully crawled, the URLs local to the same host are placed directly back into the crawler queue while the “foreign” URLs are routed over pastry by their second level domain name, ensuring that each node is responsible for a consistent subset of the URL namespace.

Each crawler queen is backed by a PageManager, which is a wrapper for a series of Berkeley DB databases.   The PageManager ensures that the crawler is entirely persistent, in the event of a force-termination the crawler will maintain its URL frontier between sessions and the list of previously crawled pages will persist.  To achieve this persistence, the PageManager uses only Transaction based key-value database as opposed to the EntityStores used by other modules of the application.  The PageManager contains four basic tables: a queue of pages to crawl, a list of pages already crawled, a list of hosts and their most recent access, and a table storing the md5 checksum of each crawled page.   The PageManager also includes a HostTracker, which keeps track of the most recent hosts that have been crawled (in a fixed-size queue) and ensures that no two drones of the same queen are ever crawling the same host.  Combined with the Pastry routing, the HostTracker ensures complete Mercator politeness.  Getting a page assignment for a drone is as simple as pulling the first entry from the Queue DB that meets the Mercator politeness requirements.  Putting a page into the DB involves storing the URL and also updating the last-access time for the appropriate host entry and storing the MD5 hash of the page contents.   Duplicate crawls are easily avoided by consulting both the list of crawled URLs and the list of content hashes.

**Indexer**:

The Indexer runs concurrently with the crawler on each node and is allocated more threads than the crawler so that page crawling does not far outpace the more computationally intensive task of page indexing. The crawled pages that are stored in the PageManager are retrieved by the Indexer, which uses the JSoup library to parse the HTML pages and extract important content. While parsing a page, the words are extracted, stemmed using a Porter Stemmer, and the other metadata information like the indices of the word, the term frequency, the weight of the word (calculated using the type of tags enclosing the word), the title of the page are extracted in order to create a URLDataObject. For each unique word, an URLDataObject is created and routed over Pastry for storage, using the stemmed root of the word to generate a hash for key-based routing. Pastry proved to be highly unstable at the message volume produced by the multithreaded indexer, so a throttling mechanism was implemented in order to regulate the rate of message flow over time under a certain ceiling. Since crawling and indexing occurred concurrently, the number of URLDataObjects awaiting transfer and storage grew rapidly and did not cease growing until program termination. A threshold was implemented on the throttle that imposed a maximum number of messages that could remain in the throttle awaiting transmission. When this threshold was exceeded, the crawling on that node would be paused while the throttle was emptied at a safe rate; crawling was resumed once the throttle was entirely empty. All the URLDataObjects that needs to be routed and the URLDataObjects that have to saved on the node were stored in the Berkeley DB by using the stemmed word as the primary key with an ArrayList of URLDataObjects as the value. This achieves an higher level of persistence of the data that needs to be sent. Even if the program crashes during the sending, it starts sending from the same point where it left off. The URLDataObjects were either appended to the entry for the word if it existed or added to a new entry, which could be extended by subsequent indexing.

**PageRank**:

For each page crawled, the crawler delivers a list of associated outbound URLs to the PageRank Manager. The PageRank manager contains only static methods and acts as a temporary local persistent store for PageRank information by appending to a file until it reaches a size appropriate for processing. Once information for 10000 URLs has been recorded, the file is closed and uploaded to Amazon S3 for later batch processing via Elastic MapReduce. The file contains a new line for each URL crawled followed by a tab-delimited list of outbound URLs contained in the page.

The actual PageRank computation is done Hadoop, a popular framework for distributed MapReduce. We used Amazon's specialized Hadoop service called Elastic MapReduce to compute the PageRank values. The input files are tab-separated files pulled from S3 after they are uploaded by the main application. The mapper program reads each line of the file, interpreting the first entry in the line as the URL to be ranked and the optional second value as the known PageRank of the URL based on the last computation. In the case where no PageRank value is known for a URL (such as the first computation on a corpus) a standard value of 1.0 is originally used. The tab-delimited list of outbound URLs follows the PageRank value in each line of the file. The PageRank is computed using the following formula as below with (d = 0.85):

Macintosh HD:private:tmp:Final Project Google Report - Google Docs.png

Two important edge cases handled by our PageRank implementation are PageRank hogs and PageRank sinks. To avoid PageRank hogs, links that point to their page of origin are removed as are links that serve an obviously navigational purpose. PageRank sinks and URLs that have not been crawled simply return their PageRank to the page that gave it to them in the previous iteration to avoid the case where a dead-end page acts as a sink in the page graph.

**PageRank - Mapper**:

The Mapper first takes in an input line and emits two kinds of an intermediate key-value pairs. One kind is the URL of a page and the list of URLs it is pointing to. The other would be all the URLs that are being pointed and their corresponding PageRank value and its parent link separated with semicolon. This allows for the PageRank to be returned to the parent URL should the page be considered a PageRank sink as described above.

**PageRank - Reducer**:

Reducer checks for the whole line to reconstruct the link Structure back. If it does not finds a whole line, it is assumed that the link was not crawled yet and so we form a new link with PageRank values as the summation of incoming values and point the parent URLs as the attached link which helps in avoiding the sink.

The java program then downloads the result files and it is parsed for the URL and Page Rank values and discards the rest of the content for that particular tuple. They are then distributed across the pastry nodes based on hostnames and it is stored in Berkeley DB for retrieval to be used by Search Engine.

**Search**:

When the user inputs a search query, the query is tokenized into a series of single word queries. For each word, a thread is spawned that queries the Pastry ring for the information related to that keyword. While waiting for the keyword threads to return information, the main thread sleeps and wakes up every 250ms or upon the receipt of results from a keyword thread. If a specific keyword thread fails three times to return any information from the Pastry ring, it is ignored and search results related to the rest of the query are returned. This ensures that the results will be returned in a timely manner even if some errors are encountered in the information retrieval process. If the thread gets the message back from pastry, it updates a local hash structure in its parent class with the data corresponding to the specific token. The message comes back as an ArrayList of URLDataObjects where the corresponding token was mentioned. In the resulting hash structure, these objects are mapped against the URL they represent. Essentially, a forward index is created of each URL that matched part of the query and which query words it contains, along with the weights associated with those words as calculated by the indexer. At this point, the threads that were spawned for each keyword are merged with the main thread and the actual search functionality begins. (Appendix: Figure 5)

The first step of the search is to aggregate the relevant URLs by the absolute number of search query words that they contain. This number is determined by the size of the ArrayList associated with the URL in the hash table structure created in the previous step. At this point three threads are launched; one for fetching the PageRank of all the URLs obtained from pastry, one for search functions, and one for fetching the shopping results from APIs such as Ebay. The thread that handles search functions will span an additional thread for each URL group formed in the previous step (grouping by number of matching keywords). This is due to the assumption that results that contain the entirety of a user’s query will be more relevant even if a document that matches a smaller subset of the query has a higher page score. Each thread will in return spawn three more threads that concurrently calculate TF/IDF (the importance of the word in the document), weight (score representing the position of the word in the document and the tags in which it lies.), and proximity (the relative positions of tokens in the document). The results of these functions will be fused as one with their scores being weighted 50%, 20%, and 30% respectively to maximize relevance. Finally, the parent thread merges the results of all child threads to create one large list of URLs sorted by score. Even at this stage, however, the documents are still sorted first by the number of words from the query that they contain. This sorted list is pushed back to the original searching thread where the scores are considered as 80% of a final score that integrates a normalized PageRank value. This list is sorted and nearly ready for display to the user. If Ebay results have been requested, they are fetched from the API, given an approximate score (which is higher if the user has used a shopping-related word) and fused into the list of results to display.   
This final list of results is formatted as XML and returned via the REST Interface, which is normally called from, and parsed by, the front-end servlet, which is visible to the user.

**Front End Interface**:

The front end user interface was provided by a single Java servlet running in a Tomcat container hosted on Amazon EC2 instance.   The servlet accepted GET requests with two parameters: the search phrase, and a boolean value indicating if the user has requested EBay results.   The servlet then opened a connection on the same machine to the REST searching interface, which returns the top 100 results for the query in XML form.  The servlet then formatted the XML into simple HTML (styled with CSS) and then executed spell checking on the query as well as a general topic search to DuckDuckGo.   DuckDuckGo results were displayed on the right hand side of the page, and in the event of a spelling error the DDG suggestions corresponded to the proposed correction rather than the original query. We fetch only the relevant searches (or similar searches) for the query that the user has passed to the program. We don’t fetch any results of the query. As the user clicks on the queries suggested, it fetches the results from our system and updates the next corresponding set of related searches from DuckDuckGo.

**Extra Credit**:

In addition to the required tasks, our search engine supports checking for duplicate page content, consideration of word context, integration of EBay and DuckDuckGo search results, and query spell checking.    Duplicate pages are identified by the MD5 hash of their entire content, which is stored locally in each crawler node.  The indexer is notified of duplicate pages under the first URL by which they were discovered, and weights the page accordingly.   During the indexing stage, extracted words were considered based on the HTML tag in which they were contained, giving higher weighting to words encountered in bold, header, title, and meta tags under the assumption that they more accurately describe the content of the page.   EBay and DuckDuckGo were integrated into search results using queries to their respective APIs.  EBay results were given an approximate score based on the body of the listing text and interleaved into the search results accordingly, while DuckDuckGo results were used to provide more search suggestions based on the RelatedTopic tags in the DDG XML file for a given query as well as a possible image result where applicable.  Spell checking performs three simple actions on each word in the query: adjacent letter swaps, inserting any single letter at any position, and removing a single letter from any position.   For a word with n characters this process generates 28n + 25 possible suggestions.  This suggestion list was checked against the UNIX dictionary file on the local machine and then all suggestions that appeared in the dictionary were sorted according to their estimated frequency in the English language (provided by a downloaded database loaded into memory at the start of each session).   Finally, the top-ranking suggestion for each word was returned to the user as the most likely spelling correction.

**Debugging:** Most of the design choices outlined above were made as a result of extensive debugging, at times completely overhauling our original plans in order to repair a single exception or error. Two main types of errors affected our original architecture in ways that we believe led to significant improvements and a deeper understanding of the material: heap space allocation errors (OutOfMemoryErrors) and disk space errors. The first occurred only in long crawling sessions, and we originally believed them to be a result of the crawler keeping too much state in memory. As a result, every data structure was removed from the crawler architecture and replaced with a persistent alternative using BerekeleyDB such as the PageManager. This added an extreme degree of persistence to our application that made any fault bearable. The second solution to the memory issues was to throttle the sending of Pastry messages in a persistent queue so that a sudden influx of information would not overflow the heap. This made the system even more persistent as well as making the performance more predictable by rate-limiting Pastry transmissions. Disk space errors arose when we observed the Berkeley DB stores occupying 10-20x more hard disk space than their content necessitated. This issue was solved by a timer that limited the rate of syncing for the Berkeley DB environment in EntityStore classes and by the use of lightweight transactional databases wherever possible. So even though our system is capable of running Crawling, Indexing and Searching altogether, our database improvements shows that the space utilization of the DB has come to the smallest possible value. It is because; we are never storing the complete webpage content into the file system. We sync the database only after we clear off the PastryThrottle, so which means we are storing only the indexed data. (So, in much less than 100MB we are able to store the data of at least 10000 pages). The transactional databases showed much better performance for random read and write actions while the EntityStore sync limiting amortized the cost of expensive database actions over time and made the performance of the system faster and, again, more predictable. The debugging of these errors occupied over 40% of all development time and the search for possible causes forced us to make our code as efficient as possible and to make the application performance seamless, deterministic, and persistent.

**Evaluation**:

**Figure 2** – The number of web pages processed per minute as a function of the number of nodes in the ring. The top line represents the crawler running alone while the bottom line represents the configuration used in production of the crawler and indexer running in parallel. Note that the difference can be attributed to the time taken by the Pastry throttle to clear off the messages (in the mean time, the crawler gets paused) that it has stored and resume the crawler back again. In both cases, the performance of the application seems to scale linearly however eventually latency issues would likely put a cap on scalability.

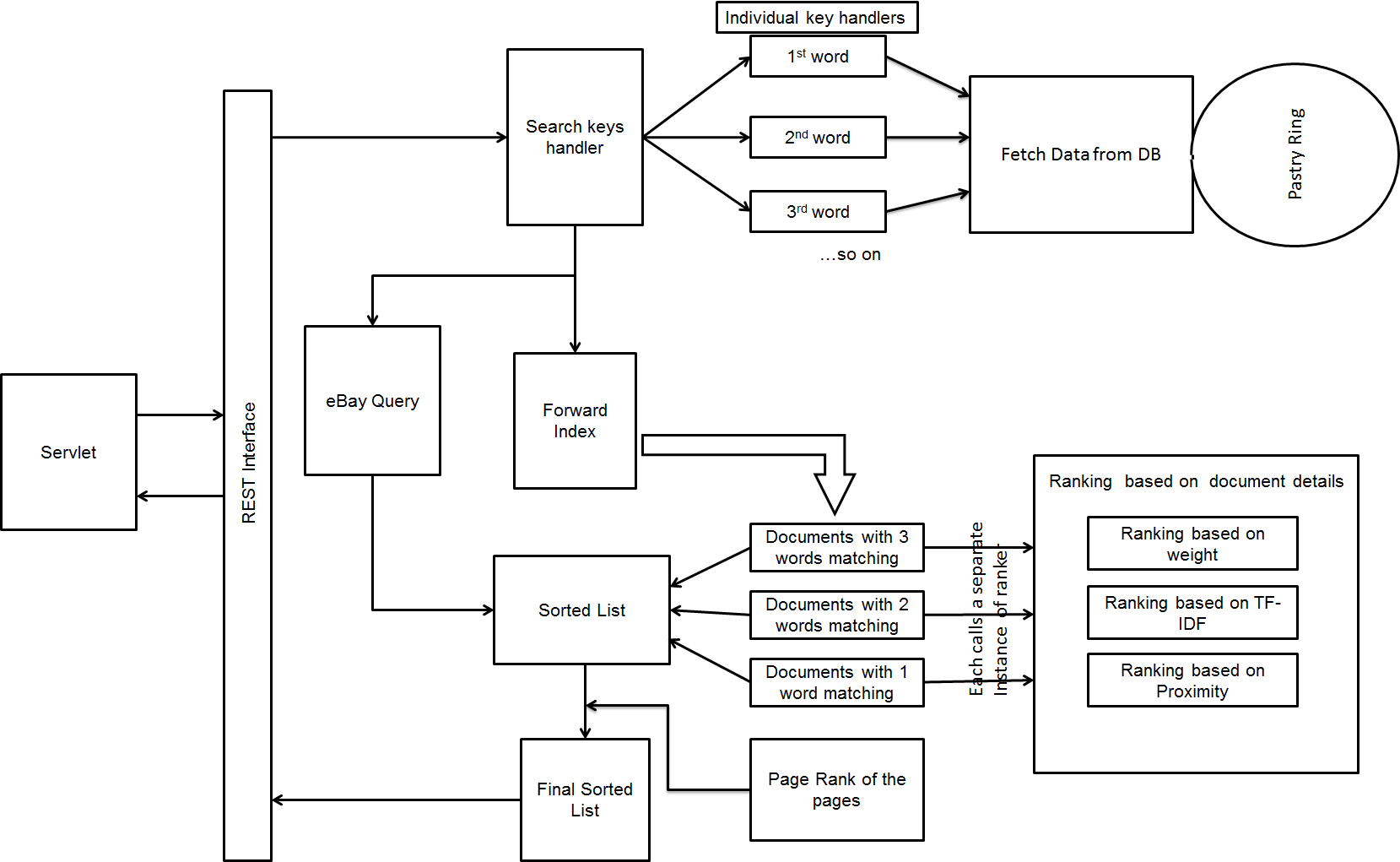
**Figure 3** – The time to process 1000 search requests delivered in concurrent groups of 100 as a function of the number of nodes in the ring. Despite the higher division of labor in higher-node configurations, the time to process search requests actually increased. We suspect this is an issue with network latency as the Pastry routing path of each message passes through more nodes as the instance count increases. It might also be the reason that on by mistake we took nodes from two different data centers and so the latency does come into actual picture here.

**Practical Testing:** The site has been live for 6 hours after the deployment. We had around 1100 hits in less than 6 hours, the system out performed our expectations and it is able to accept a huge amount of load and also a variety of search queries from different users without breaking the system. One notable point about the search capacity can be checked by doing a query for the stop words. The OutOfMemoryError will give out the result for any query in less than 900ms.

**Conclusions**:  
 The successes described throughout this report were the result of days of error hunting and bug fixing.   Our original plans for the search engine included a much-simplified architecture that placed a heavy load on Pastry and kept the bulk of the crawler state in memory.  We found that our originally provisions for scalability and persistence were not nearly sufficient to complete our goals.  However, we soon discovered that operating multithreaded programs on a distributed system will reveal bugs that have no bearing on programs of a smaller scale.  The search engine takes the name “OutOfMemoryError” from the exception that plagued the first week of development, cropping up in all phases of operation.   The result of troubleshooting this error was the creation of an application that is incredibly fault tolerant and entirely persistent.   As long as the database files are not corrupted the system can be closed and restarted at any time without any loss of state or negative performance effects.  The errors we discovered forced us to implement a system that was robust far beyond our needs and extremely easy to scale and maintain in any environment. It is also because of a smaller DB size, our search functionality has a high throughput value and also because of highly multithreaded system, the search engine is able to give out the results in less than 900ms. We believe that all of our initial goals were achieved by our application and exceeded by the highly fault tolerant nature of the final design.

**Appendix**:

**Diagram of Search Architecture**:



**Figure 5-Search Architecture**