Group Name: Timedex (http://www.timedex.org)
Team Members: Alex Loddengaard, Brandon Bell, Robert Gay, Sean McCarthy

Project Goals

Our high-level goal in creating Timedex was to extract events from a full-size dump of the Wikipedia corpus and then create an easy-to-use web interface around the events we extracted. The main goal of the web interface was to make the events we extracted accessible in fast and intuitive way.

System Architecture

The most important component of our architecture is the database that stores the Wikipedia dump and the extracted events. This database is common to our event extraction step and the web application and contains imported Wikipedia tables containing pages, revisions, text, and page links as well as an additional table that stores events. Figure 1 gives an overview of the system architecture.

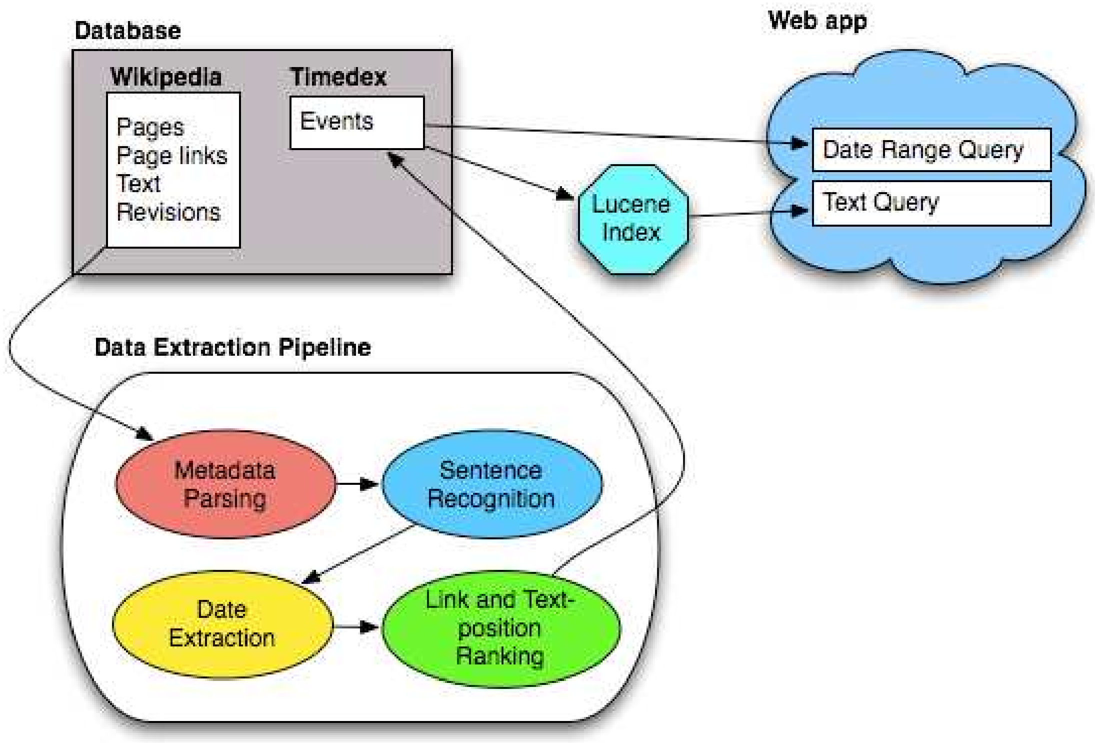


Figure 1: Timedex system architecture.

A Hibernate persistence/data-access layer is configured around the database in order to allow the web app and the extractor to use the same code to access the database. The data-access layer consists of a set of classes that are mapped to database tables by Hibernate. Because of the Hibernate API, the data-access code doesn't require a single SQL statement, making it maintainable and robust. We were also able to write effective JUnit tests for all of our data-access code.

Extraction
The goal of the extraction step is to fill the events table in the database. To do this, we first scan through the text of each Wikipedia article in the default namespace and parse the page text into a tree of headings, with the page title as the root heading. (We ignore all non-heading-related Wikipedia formatting.) Each heading in the tree has the text under it organized into a list of sentences. For sentence edge detection, we used a library called Lingpipe that works well in almost all cases. We have yet to observe a mistake in sentence parsing by our system on real data; it has only messed up on some carefully chosen test cases (e.g., sentences that end with abbreviations like 'B.C.' or 'etc.').

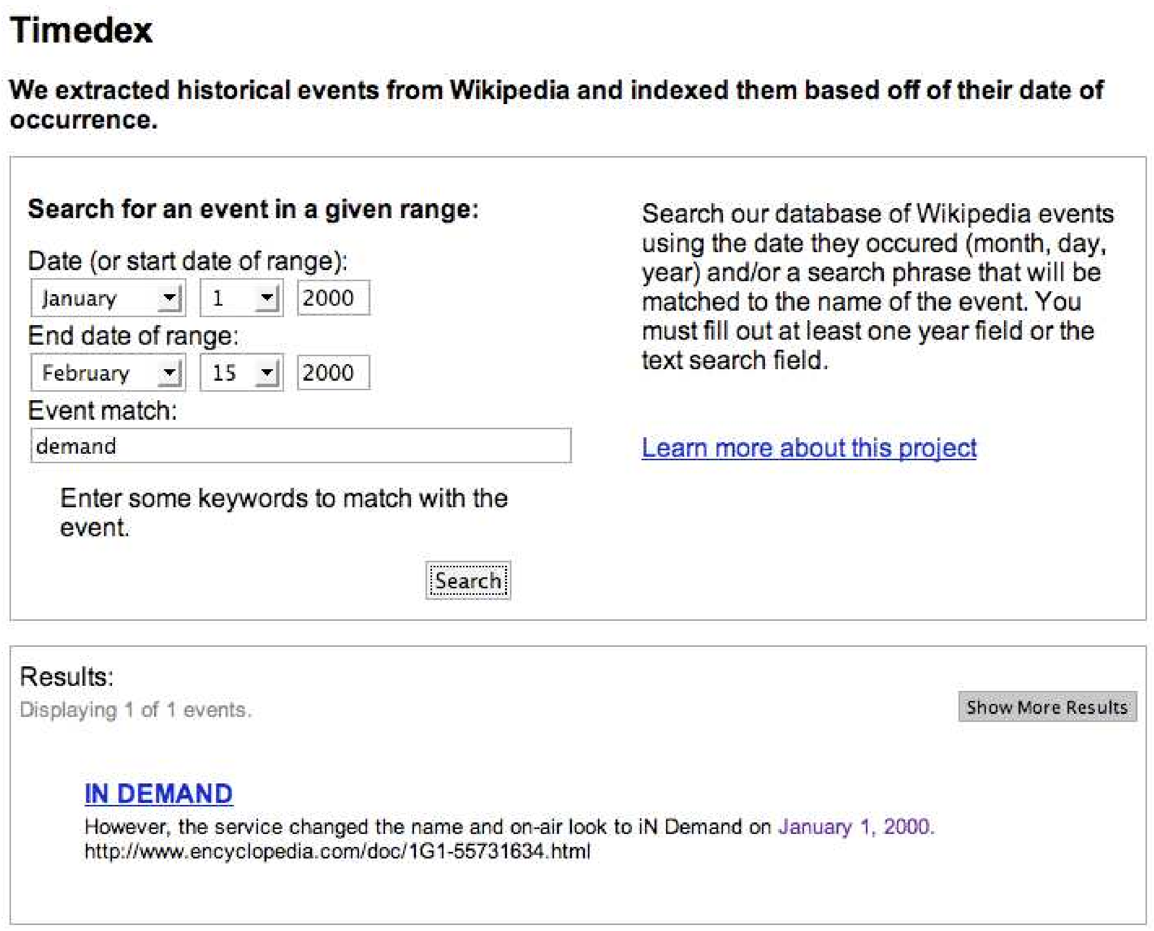
Next, we try to detect dates in each sentence. This is done with a simple set of regular expressions. Although there are many different ways to format dates, we chose to focus on the types of dates that might be encountered in the text of a general Wikipedia article. These formats are the ones outlined in Wikipedia's style guidelines. Since many dates are only years without indicated eras like BC or AD, we decided to treat all numbers between 1500 and 3000 as dates if they are preceded by certain keywords such as "in" or "before." Although some of these decisions may omit some date formats, we chose to use this strict date detection to increase the precision of our output. Since dates are easy to detect in general, we have found that the results find all or almost all dates in the average article with few false positives. Our precision and recall are good enough that it would be difficult to collect enough manually processed data to nail down those numbers. Sentences that have a date are stored in the events database along with their parent headings, page title, extracted date, and number of links into the page. One thing we have avoided is extracting more than one date from a given sentence. We made a conscious decision in the early planning stages that it was not worth worrying about that.

The date extraction matches dates that are less specific than a given day. (e.g., "May 2007"). In order to support this in the database, we have two columns: a start date and an end date. When we match something like "May 2007" (which is accurate to the month) we set the start date to be the first day of that month in that year and the end date to be the last day of that month in that year. In this way we are able to take advantage of the database's date type to allow simple, efficient searching and are still able to differentiate between events where we have all the date information and events where the date information is incomplete. We implanted support for ranges over a month and over a year, but it would not be difficult to extend the system to, for example, treat "the 13th century" as ranging from the beginning of 1200 to the end of 1299.

The final step of extraction is the creation of a Lucene index. The Lucene index stores three pieces of information: an event sentence, the subheading the event was in, and the id of the event, where the sentence and the subheadings are indexed. Indexing the sentence and the subheadings allows for robust keyword searches in the web app. Although the Lucene library handles most of the details, the index is created treating each sentence combined with the headings it is under as a separate document. Each document is separated into words, which are the basis for the index. This means that the results for a query are composed of a list of the sentences that match the query. Because the sentences are referenced by their id in the database, these Lucene queries can be easily combined with date range queries on the database. The index is stored in memory on our production machine for better performance, although we used a disk-based index during development.

Web Application

The user interface for the web app is comprised of a date range search and a keyword search. Figure 2 shows a screenshot of the Timedex web app (also accessible at http://www.timedex.org).



Timedex

We extracted historical events from Wikipedia and indexed them based off of their date of occurrence.

Search for an event in a given range:

Date (or start date of range): January 1 2000

End date of range: February 15 2000

Event match: demand

Enter some keywords to match with the event.

Search our database of Wikipedia events using the date they occured (month, day, year) and/or a search phrase that will be matched to the name of the event. You must fill out at least one year field or the text search field.

Learn more about this project

Results:

Displaying 1 of 1 events.

IN DEMAND

However, the service changed the name and on-air look to iN Demand on January 1, 2000.

http://www.encyclopedia.com/doc/1G1-55731634.html

Figure 2: Screenshot of the Timedex UI.

The date range search runs a query against the events table and returns all the events within the given range. The query also sorts by rank, allowing us to limit the number of results returned without potentially losing good events. Searching by keywords looks up events using the Lucene index. When both the range search and the keyword search are used, an intersection between both searches is computed. Figure 3 breaks down the way in which queries work.

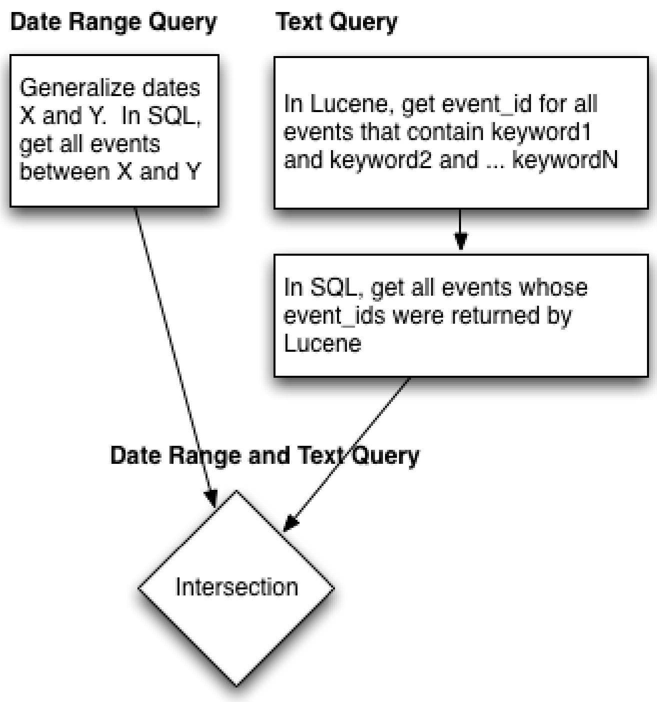

Date Range Query
Text Query
Generalize dates X and Y. In SQL, get all events between X and Y
In Lucene, get event_id for all events that contain keyword1 and keyword2 and ... keywordN
In SQL, get all events whose event_ids were returned by Lucene
Date Range and Text Query
Intersection

Figure 3: Timedex query breakdown.

Figure 4 shows a breakdown of what each event is comprised of along with metadata about the event such as its rank.

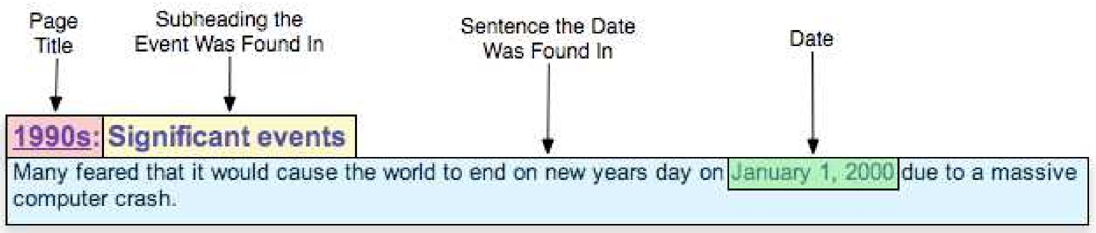

Page Title
Subheading the Event Was Found In
Sentence the Date Was Found In
Date
1990s:
Significant events
Many feared that it would cause the world to end on new years day on January 1, 2000 due to a massive computer crash.

Figure 4: Result breakdown.

The presentation layer uses Spring MVC, a Java web framework, for server-side dependency injection and mapping request URLs to specific controllers.

Workflow

When a user hits the “Submit” button, an asynchronous JavaScript request is made to a query controller. The request contains all of the search parameters such as the start month, the end day, and the keywords. The controller returns JavaScript Object Notation (JSON) plain text that is interpreted by the JavaScript as a sequence of serialized objects. These objects are then displayed and can be either shown or hidden depending on their rank and their positions in the distribution of ranks in the result set. Figure 5 shows the steps that happen when a user submits a query.

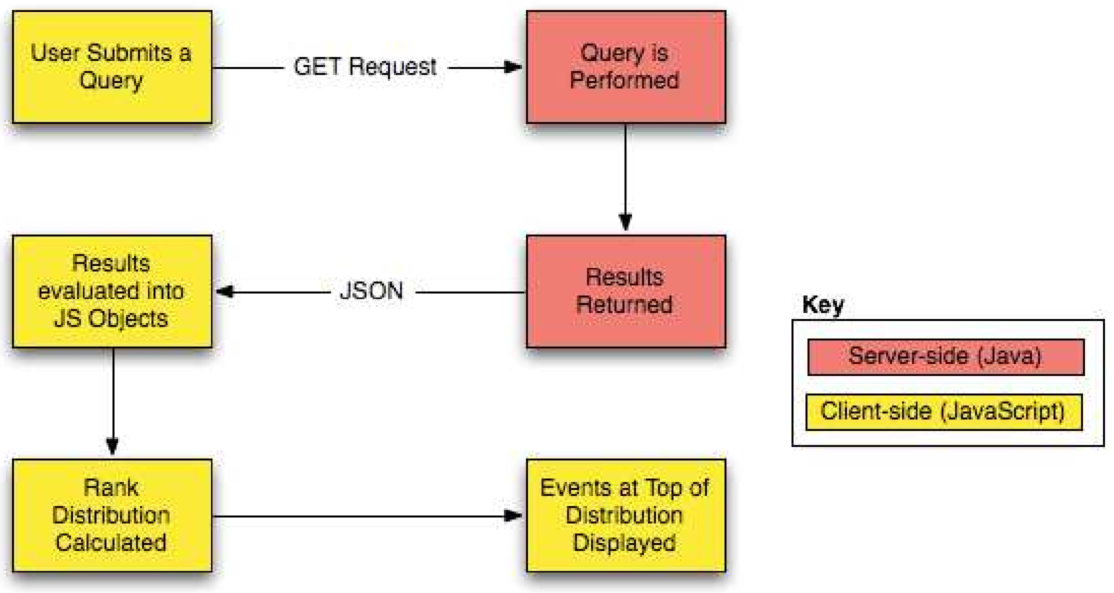


Figure 5: Web app workflow.

Database Challenges

One large challenge encountered was the management of the Wikipedia corpus. The entire corpus exceeded 10GB, including 177 million rows in the page links table and 5 million rows in the page table. The production web server only had 4GB of main memory, so all of our applications had to be careful to avoid excessive memory page thrashing.

Another challenge became apparent with the use of a naïve database transaction-management strategy. Initially the transaction strategy was implemented as one transaction per Wikipedia page processed. This strategy worked, but it was found that running this algorithm on a dataset with 5,000 pages took around 3.5 minutes even though the entire database could fit in memory and therefore, in theory, be processed all at once. This translated into a huge delay on the production machine. The solution to this problem was to refactor the data-access layer to allow multiple select and update statements per transaction. This allowed partitioning the set of pages up into large chunks that could fit into memory and be processed as a whole. In the case of the small dataset, the total number of pages was much smaller than the chunk size, so the improvement was very dramatic: the time to run the algorithm dropped to around 48 seconds, a drastic improvement. Although this new transaction model did improve performance on the large dataset, the time that the PageRank algorithm was taking was still far too long.

After not finding large enough gains in performance by implementing a better transaction strategy, a third implementation of PageRank was written that was multi-threaded and processed multiple chunks at a time. The idea behind this implementation was that the Java process would fill the web server’s main memory with data and only access the database when a chunk was finished being processed. Unfortunately this didn’t account for disk seeks when a query is performed in MySQL, so the algorithm still ran too slowly to be completed during the course of the quarter. Our solution to our PageRank problems was ultimately to abandon PageRank altogether for the purposes of this class. The ranking algorithm instead just counts the number of links coming into each page, and uses that in combination with the depth of the event's sentence in the page's subheading hierarchy to calculate a relative rank.

A transaction model similar to the one used for PageRank was also used for event extraction. Event extraction took a matter of seconds to execute on a 5,000-page data set and on the order of a few days for the full Wikipedia corpus. The total number of events ended up being around 7.5 million, implying there are roughly 1.5 events associated with each page on Wikipedia.

Issues With Extraction
Our original plan for event extraction involved constructing a summary of the event from important words in the sentence. Only this summary and the date itself would then be displayed to the user. In pursuit of this goal, we set up an intermediate step to tag the parts of speech in our sentences using Lingpipe and also constructed a Mallet CRF to process these features for us. A simple UI was written to allow us to label some training data quickly and efficiently.

Unfortunately, what we discovered when running the training UI was that most of the sentences were unsalvageable. Problems included frequent pronoun subjects, events spread over more than one sentence, and dates just mentioned in passing that had almost no relation to the content of the sentence. We knew going into the project that "understanding" any given sentence enough to edit it was going to be hard and we quickly realized processing one sentence at a time was not likely to result in data that was useful to the classifier. Even if the classifier were able to extract important words from the sentence, the summary that we would display would be next to useless because of extensive use of pronouns. Generating good training data also turned out to be extremely difficult because even *we* had trouble figuring out what the relevant parts of the sentence were. Solving this problem would be a great machine learning project but it is well beyond the scope of what we had planned for.

Luckily, we also noticed while using the training UI that the closest heading was a surprisingly good indicator for the subject or description of the event. Since we were already tracking that information, it was easy to switch over to a model where the main piece of text representing an event was the heading of the section of the parent Wikipedia page. We were quite disappointed that we did not end up using any machine learning to solve the problem, but are very happy with the result as a whole.

Thoughts on Storing Data in a Database Versus Files
The original reason behind managing data in MySQL was because it would allow us to make our data accessible to a web app very easily. At the time of this decision, we were not considering writing a PageRank algorithm or any other computationally intensive algorithm. As previously illustrated, algorithms such as PageRank do not perform well in

a single MySQL database. Had we made the decision from the beginning to use custom file formats and Hadoop, we could have been computed PageRank much more efficiently. We would have had to write custom scripts to turn plain text files into SQL insert statements, but that seems like a small price to pay for a much more robust, scalable computational model. MySQL is great at allowing for easy inserts and queries, but complex algorithms such as PageRank should be ran in a distributed environment with a custom file structure.

Conclusions

On the whole, our group is very pleased with the outcome of this project. We had initially hoped to use more machine learning in our approach, and in hindsight should have realized sooner how complex our efforts would have to be for that to work. We also had hoped to have more time to work on a graphical timeline UI, but in the end it was more important to us to provide meaningful data so we focused on that. We think that the end result is interesting and quite useful, and we are proud of what we were able to accomplish in only a few weeks.

Breakdown of work:

The following people did the following things:

- Alex
  - Data-access layer, database work, JavaScript/front-end work, attempted PageRank
- Brandon
  - Date extraction, Lucene index integration, event detection using CRF (unused), exploration of CRF/HMM capabilities of Mallet
- Sean
  - Wikipedia page text parsing, events table schema design, ranking results, part-of-speech labeling for CRF (unused), training data labeling UI for CRF (unused), result display
- Robert
  - Events table schema design, web app setup and maintenance, transaction performance tuning, sentence extraction application, front-end JavaScript and result display

External Libraries and Tools Used

The following external libraries were used in different areas:

- Mallet
    - Machine learning API. Used for CRF (unused).
- Lingpipe
    - Sentence recognition, part-of-speech labeling (unused).
- Lucene
    - Full-text index of the extracted events.
- Hibernate
    - Relational/persistence API for Java.
- Spring MVC
    - Java MVC framework.
- Scriptaculous
    - JavaScript library for visual effects.
- Maven
    - A Java build tool
- JUnit
    - Java unit testing framework

Viewing the Project

Visit http://www.timedex.org in any browser, although only Firefox has been tested.