ENABLING VERTICAL SEARCH OVER WIKIPEDIA

BY

STEVEN KUO CHIENG

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Kevin Chen-Chuan Chang
The massive amount of information on the Web has led to the proliferation of vertical search engines, which are specialized for specific domains. Such engines can offer superior results through gathering and utilizing domain-specific information. In this thesis, we explore the idea of applying similar techniques to Wikipedia. We construct a conceptual model of vertical search and explain why Wikipedia is especially suitable for this form of search. We go on to analyze the difficulties of making full use of Wikipedia’s advantages, and offer possible solutions. These solutions were tested through implementation of simple search scenarios across two distinct database architectures. Finally, we compare the performance and ease of implementation of the two architectures, consider the quality of results obtained from our searches, and offer suggestions for future work.
# TABLE OF CONTENTS

1  INTRODUCTION ................................................. 1
2  VERTICAL SEARCH - REQUIREMENTS AND SCENARIOS .... 4
3  WIKIPEDIA: BENEFITS AND REQUIREMENTS .............. 8
4  DATABASE SELECTION AND SCHEMA ..................... 17
5  IMPLEMENTATION AND PERFORMANCE ................... 20
6  RELATED WORK ............................................. 29
7  FUTURE WORK ............................................... 30
8  CONCLUSION ................................................. 32
REFERENCES ................................................... 33
Over the past decades, the Web has grown into a massive source of information, covering every field and every format. With this increase in size comes a corresponding need for effective search, which separates the information desired by the user from the vast quantities of information that is irrelevant or useless for his purposes. The majority of queries over the Web are handled by general purpose search engines such as Google (www.google.com) and Bing (www.bing.com) with satisfactory results, but the same generality which allows them to handle all queries implies a lack of depth within specific domains or topics. This has given rise to numerous vertical search engines, which are tailored for specific domains or verticals, such as consumer products, companies, videos, or any other category in which it is possible for a specialized search engine to offer superior results than a general search engine, at the cost of being incapable of servicing queries that lie outside their domain. We assert that the benefits of vertical search are not limited to the Web, but also apply to Wikipedia (www.wikipedia.com), a massive, user-edited encyclopedia with over 3 million English articles [1]. The encyclopedia covers all manner of subjects and topics, and every article can be edited by any user. As a result, the information which is available on the Web has, over time, accumulated within Wikipedia. Like the Web, coverage of topics within Wikipedia is far beyond that of conventional encyclopedias [2]. But unlike the Web, data in Wikipedia is generally correct, with accuracy on major topics comparable to that of respected encyclopedias such as the Encyclopaedia Britannica [3]. Because of these factors, one may often turn to Wikipedia for information instead of, or in addition to, the rest of the Web. Therefore, improving the quality of search over Wikipedia is a worthy goal. At present, search on Wikipedia is limited to a general purpose search, which is incapable of executing advanced queries. In order to give weight to our assertion that this should be accomplished through vertical search, it is necessary to first discuss how such engines operate.
The power of vertical search engines comes from two sources: the additional, domain-specific information which they store on the pages that they search over, and the search algorithms which utilize this additional information to produce more relevant results. Of these two comparative advantages, the former is a much larger obstacle to the creation of a vertical search engine. Given a hypothetical database with all the information on the Internet for a specific domain neatly laid out in some desired format, it would not take much imagination to come up with efficient ways of querying this information. However, the task of constructing such a database is not simple. Since the scope of the Internet is far too large for this to be achieved by human effort, the great majority of this work must be performed through an automated software agent. However, it is not clear how such agents should go about gathering data. Information on the Web follows no particular structure and nor are pages, in general, laid out in any fixed format. Whatever formatting or structure exists is primarily for the benefit of a human reader. Terms can be ambiguous, and apply across many domains; e.g., the term Honda may refer to a car company, to one of several businessmen, or to a knot, in addition to other possibilities.

One solution to these issues is to restrict the sites covered by the vertical search engine to those which have a well understood semantic structure. Instead of searching for consumer product information across all pages on the web, one might instead only search the sites of a few retailers, which follow a consistent format. Then it is possible to safely retrieve information with confidence, knowing that information in this location on a page must refer to price, while a string in another location must refer to the brand, and formerly ambiguous terms can be assumed to be in the context of the shopping and product domain instead of, for instance, the context of political assassinations, which is out of place on a retail site. The drawback of this approach is that now we are no longer taking advantage of the vast wealth of information freely available on the Internet instead of drawing from a huge quantity of badly formatted data, we are now only drawing from a small quantity of well formatted data. The ideal situation would be one in which all pages are well formatted and the information contained in those pages is organized as so to be easily understood by software agents. This ideal, known as the Semantic Web, would give us the best of both worlds the ability to gather information from a huge quantity of well formatted data [4].
Unfortunately, most of the pages on the Web are not in such a state, and there are various obstacles, such as the necessity of standardizing ontologies and additional overhead in page creation, which make it questionable whether the Semantic Web will ever be a reality [5] whichsemantic. As a result, vertical search across the Web is likely to always face difficulty in acquiring domain-specific information with both good accuracy and good coverage.

Wikipedia pages possess a much greater degree of semantic structure than pages on the Web, and so come closer to the ideal of the Semantic Web. Therefore it is logical to investigate vertical search over Wikipedia, as the same considerations that motivate vertical search over the Web are also present for Wikipedia, but the primary obstacles in the way of gathering data are significantly reduced. However, they are not eliminated, and taking full advantage of Wikipedia’s semantic structure brings up problems of its own. We will later examine these problems and their solutions. In summary, we seek to identify why vertical search over Wikipedia is desirable, the problems involved in implementation, and solutions to those problems. We then demonstrate a few examples of vertical search over Wikipedia as a proof of concept.

The rest of this thesis is organized as follows. In chapter 2, we propose a model of general and vertical search, and provide examples of vertical engines. In chapter 3, we identify the key benefits of Wikipedia over the Web with regards to vertical search, the technical problems involved in utilizing these advantages, and how those problems may be addressed. In chapter 4, we select two different databases for this task on the basis of their ability to meet the problems identified in chapter 3, and explain the schema chosen for each database. In chapter 5, we describe our implementations of vertical search across both databases and compare their performance. In chapter 6, we offer a brief survey of related work, and in chapter 7, we discuss remaining issues and future work. Finally, we conclude in chapter 8.
Search, as a whole, has always been about solving the proverbial "needle in a haystack" problem of identifying a relatively small amount of key, relevant documents, or even especially relevant portions of those documents, out of a much larger corpus of documents. The ultimate judge of relevance is the user, but the size of the corpus is such that it is impractical or impossible to do this manually. Therefore, automated methods are employed in order to present the user with a much smaller set of documents, which are either relevant or which contain the relevant documents.

A general search engine, having no understanding of the information it is searching over, is forced to use methods which are applicable across all domains. Generally, this will take the form of a keyword search, employing a user query which consists of terms which he believes documents relevant to his needs will contain. There are many refinements of this process, such as allowing the user to specify various weights, stemming, or the use of additional information in the search, such as the frequency with which the document is visited or referenced. The key factor is that these methods do not require the engine to understand the content of the page, and so apply across all domains. A vertical search engine, on the other hand, will be able to understand that portion of the documents it searches over which correspond to its domain. Thus, a search engine for the book domain would be able to recognize the presence of books in the documents that it searches over, while engines on apartments, consumer products, automobiles, celebrities, etc. would be able to recognize and parse out information relevant to their topics. This information in turn leads to more focused or relevant queries, because the attributes that a vertical engine can query over are both more plentiful and more useful.

We can now make a simple model of search. We begin with a set of documents, \( S \), where each document holds information encoded in some format. This information can be grouped together into specific entities or objects,
which compromise another set, $E$. Each entity belongs to some data type, $T$, which describes what kind of entity it is and hence what kind of information would exist for it. Each piece of descriptive information is an attribute, and each data type maps to a set of attributes, $A$. These entities and attributes are encoded within the content of the documents which are searched over. Finally, a domain, $D$, represents some concept or topic which a user may be interested, and each entity corresponds to one or more domains. A query on a domain is an operation or algorithm which searches over the set of entities which fall into that domain and returns those entities or attributes which are relevant. The query may take as parameters attribute values, which allow the user to specify to the engine what he is looking for.

This allows us to state more clearly the key differences between a general search and a vertical search. A general search is a specific instance of a vertical search which creates a one to one mapping between every document and an entity which represents that document, which in turn makes up the domain of the general search. The entity’s attributes are those which are common to all documents, which may be title, author, text, etc., and the queries that can be performed by this general search make use of these common attributes and thus are limited. The entities which it returns are in fact the documents which make up the collection, and it is left to the user to read through these documents and discover whatever actual entities and attributes he was interested in. A vertical search engine, on the other hand, is first determined by the domain it is searching over and thus what entities it is interested in. It parses the documents of its corpus in order to extract from them a set of entities and their attributes, where each entity falls under the domain which the engine is concerned with. It then searches over these entities, making use of attributes which are more relevant to its domain than the universal attributes employed by general search engines.

In the context of the Web, the collection of documents is the Web itself, while each document is a page on the web. The domain employed by a general search engine is the domain of Web page entities, and the attributes of these entities are those pieces of information relevant to Web pages such as URL, domain, title, text, number of visits, importance as defined by PageRank [6], and so on. A vertical search engine is one which extracts from these pages those entities which fall within its domain. It examines the content of each page, parsing out the existence of these entities and the attributes
which describe these entities. Both engines will query over their respective entities, making use of the relevant attributes.

Let us now end with some clear examples of vertical search, beginning with the book domain. We previously stated that a book vertical search engine would be capable of recognizing books and attributes of books within the corpus it searches over. We can now see that the practical result of this is to translate that corpus from a set of documents to a set of book entities, each of which possesses a set of attributes which describe it. These attributes would be items like title, author, publisher, genre, date of publication, ISBN, and so on. We can make focused queries on this set of books, such as “return all books where the author’s name contains Smith”. If we were to attempt this same query using a general search engine, we might try something like “books author Smith”, which would search for documents where the text contained these keywords, but it is obvious that this is a very flawed solution. For example, a document that contains information about a book need not include any form of the word ”books”, for much the same reason why the home pages of search engines do not contain the term ”search engine”, and the same principle applies to the term ”author”. Even if it contains these keywords, they do not necessarily have anything to do with each other; a document may talk about a scandal involving the politician Smith in the first paragraph and then criticize an author’s books in another. Furthermore, we can see that a paragraph that reviews a biography of John Smith may contain all three terms in close proximity, yet fail to match the query because we are looking for ”books by Smith” and not ”books on Smith”. Finally, the results are not in the most desirable format. Even with perfect precision and recall, we would still have to sort through the results in order to find the information that we want. Information on any given book would be split up across multiple documents, which we must read through in order to manually piece together each book’s data, while each book entity returned by the vertical engine would come with all its attributes attached.

Another example would operate over the human domain, and makes use of the link structure of the documents. Here, we create a set of persons, where the attributes include a list of links to other people. By following these links, we can find, for any given person, all people who he is connected to, and by following links from those people we can find people of varying distances from a person, e.g. “all people within a distance of 3 from Warren Buffett”.

6
It is interesting to be able to find such networks or groups, and observe, for example, how there may be only a short distance between two people who seem completely unrelated. Another application might be to determine the distance between two people, allowing the user to see how many degrees of separation lie between them. With a general search engine, we can determine the distance between documents, but not the distance between the people mentioned in them.

Finally, we present an example in which we operate over multiple types of entities that feature some kind of relationship. We search over the music domain, which for our purposes consists of musical artists, including bands or other groups of musical artists, and albums. Our query simply consists of the name of the musical artist which we wish to find information from. Under a general search engine, this would give us a set of documents which contain information about that artist, but we would have to search through those documents to actually find the information. Here, we could just directly present all the information about that artist, but we can go one step further and point out additional information to the user. For example, if we are looking for information about a band, we can look for other bands which include or once included members of the band, by obtaining the list of members of the original band and then searching across the members attribute of all bands to see whether they include one of those members. Another possibility is to look for albums which are created by that band, and to return some information about them in the search results as well. This example demonstrates how a vertical search engine can automatically find entities that are related to the ones explicitly provided by the user, and how information about an entity is often present in the attributes of other entities.
The general structure of the Web is well known, and we so we will de-
scribe it only to the extent necessary to make an informed comparison with
Wikipedia’s structure. In general, it is composed of a collection of pages,
where each page consists of a document storing text and other data in no
particular format or structure. These pages are each associated with a unique
identifier called a URL, and often pages are grouped together into sites based
on shared portions of their URLs. Pages also often contain links, which are
references to the URLs of other pages, and their presence gives the Web a
graph structure, in which each node is a page and each edge is a link. The in-
formation present in the Web is located in both the data of each page and in
the links between them, and it is this information which search engines, both
general and vertical, attempt to utilize. As discussed in the introduction,
a general search engine can use the text as-is for searches, while extracting
additional information out of the text is a great challenge for a vertical search
engine, which must be overcome if it is to demonstrate superior results even
within a narrow domain of queries.

The structure of Wikipedia is similar to that of the Web. It consists of
a collection of pages, which hold text and other data, usually images. Each
page contains links, both to other Wikipedia pages and also to pages on the
Web, and so the structure of Wikipedia can also be represented as a graph.
However, the link structure of Wikipedia is denser than that of the Web,
and links generally have more relevancy [7]. Each page on Wikipedia can
belong to categories, which group together pages that are related in some
way, and this is analogous to the grouping of Web pages into sites. So far,
we do not see any huge difference between Wikipedia and the Web, and in
fact it is in content that Wikipedia pages and Web pages primarily differ.
Each Wikipedia page is an article on a topic, and most or all information
in Wikipedia on that topic is concentrated onto that one page, which is
identified by its title the topic that it covers. In contrast, information on
the Web for a topic is invariably split between many separate pages. We
can also state that each Wikipedia page maps to one entity, while each Web
page tends to contain information for multiple entities, and each entity’s
information is split between multiple pages. This means that it is easy to
fetch information for an entity on Wikipedia and to map pages to entities,
while on the Web these are difficult and time consuming tasks.

Most importantly, Wikipedia pages often have a certain format for some
of their data. Pages often contain a set of key-value attributes, called an
infobox, in which various information is displayed in a structured format. It
is easy for a software agent to both obtain the list of keys and also to obtain
the value for each key. Each page could, in principle, have an independent
infobox, but generally pages of similar content will also share infoboxes. As
such, Wikipedia pages on country entities will tend to have the country in-
foobox, which contains keys and values that are relevant to countries, such
as gross domestic product and population. This acts both as a signal of the
content of a page, showing what entity it covers, and also as a way to imme-
diately find domain specific or entity specific information. Naturally, since
each country is different, the content of each infobox may also be different,
but if you have a set of pages with country infoboxes, it is likely that for each
page, the current population of that country will be a value in the infobox
and the key will be a certain string that is constant across each page. In
general, different instances of the same infobox will share the same common
attributes, which are the ones most useful to query on.

We previously identified two main factors in vertical search: obtaining
domain-specific information, and utilizing that information in searches. Of
these two, we concluded that the first task is difficult, while the second task is
relatively easy. The primary advantage of Wikipedia over the Web in terms
of vertical search comes from easing the difficulty of the first task. Because
information for each entity is concentrated onto a single page, it is easier
to find the relevant page or pages, and because pages have more semantic
structure, it is easier to extract domain-specific information. If we want to
find all information on books that Wikipedia has to offer, we can start by
simply looking for all pages with book infoboxes. The structured format of
the infoboxes allows us to easily obtain common book attributes: if we want
to obtain the author of a book, we can simply look at the author field of the
book’s infobox. In contrast, on the Web the information for any given book
is spread across numerous pages, and the key information such as author or
date of publication is difficult for a software agent to extract.

Given that we can successfully extract such information from infoboxes, we
assert that the main factor in implementing vertical search over Wikipedia
is in successfully storing the information so that it can be efficiently queried.
This translates to a question of how to store Wikipedia pages within a
database. There are four main issues that must be satisfactorily addressed
in order to do this.

**Issue 1: Large corpus size.** The size of Wikipedia is very large: as of
November 25, 2010, there are 3,485,971 English articles. Therefore, if we are
to store the entirety of Wikipedia, we must use a database which is capable
of quickly searching over a large number of records, and we must restrict
ourselves to queries which can be completed in a reasonable time frame. The
former is not difficult for almost all major databases, but in the latter case,
the speed of the database may effect what kinds of queries we can utilize.
For example, some operations, such as regular expression matching, are time
consuming, especially if they are performed over a large number of strings,
and it may be that only practical testing can determine which databases can
get away with what types of queries, and under which situations. We must
also consider that each vertical search engine has its own unique queries,
which will most likely differ significantly from the queries of other vertical
ingines. If each engine’s queries require their own specialized indexes to
be built, then the large size of the corpus implies indexes of corresponding
size, resulting in space problems (indexes that do not fit in memory), time
problems (queries that take too long to run), or both.

If we suppose that a vertical search engine’s domain includes only a small
subset of Wikipedia’s pages, which is quite likely as the number of entity
types involved in a domain is usually small, then we can simply store only
those pages. Thus, the size of the dataset we must deal with decreases greatly.
However, since Wikipedia contains so much information, even a small subset
of its data may be considerable in size and deserve some consideration for
performance issues. We acknowledge that no matter how large the size of
Wikipedia is or becomes, the Web as a whole is much larger in size and
that vertical search over the Web has already been implemented with some
success. However, as previously stated, most vertical search over the Web
actually utilizes a small fraction of its pages, and that subset may be smaller
in size than Wikipedia, so this does not mean that Wikipedia’s size is a nonissue. Furthermore, it may take impractical amounts of time to query even a tiny set of documents if the query is arbitrarily complex, and the larger the set of documents, the less time-consuming the types of queries that can be applied to it for any given upper bound in query time. Since vertical search engines may have justification for complicated queries that take full advantage of their domain-specific information, it is sensible to seek a database that is fast enough to make such queries practical.

Finally, even if the database is extremely fast, if the way that the database stores information is not compatible with how vertical search entities should be organized, or if the data is difficult to query, then searches will suffer in response time. A large corpus size will magnify whatever performance penalties are incurred by this incompatibility. Even if a convoluted method of querying data, made necessary by some quirk of data storage, is orders of magnitude slower than straightforward queries over a compatible database, the difference is hidden until the number of pages searched over becomes sufficiently large - the difference between 0.001 seconds and 0.1 seconds is insignificant, but the difference between a 1 second query and a 100 second query is large. Therefore, Wikipedia’s large corpus size can act to make other issues more significant, if there are ways to solve those issues by trading off speed or performance.

Issue 2: Entity resolution. In Wikipedia, each page is on some topic, and thus represents the entity indicated by the topic’s name. The most straightforward way to determine what kind of entity a page represents is by looking at the page’s infobox type, which is generally quite clear as to the matter, e.g. if the infobox type is ”book”, then the page is certain to represent a book, and if we wish to search across the domain of books then we only have to handle pages which possess that infobox type. However, there are two problems with this. One is that pages that do not contain infoboxes cannot be mapped to any entity type using this method, and it is not clear how it can be done in the absence of an infobox. The categories a page is present within provides some clues, but page categorization on Wikipedia is both inconsistent and incomplete, and we know of no way to connect these categories with infobox types. Since we can only query across those pages which map to entities, it follows that we can only query across those pages with infoboxes.
Second, while every page has only one infobox type, and every page is a single entity, it does not follow that every page conceptually only maps to one entity. What we mean by this is that if we are searching across the domain of NBA players, then we will have no problems with the page titled "Michael Jordan" as the infobox type for that page is "NBA Player". However, if we wish to search across the domain of all basketball players, then we must handle not only handle "NBA Player" pages, but also pages where the infobox type corresponds to other types of basketball players. This in turn is a subset of sports players, which in turn is a subset of people. So we see that there is an organization between entities and thus the infoboxes which represent them, and if we want to search on all people, it is not trivial to figure out all the infobox types which correspond to people. In addition, even if we figure this out, we do not know if tomorrow someone will add a new infobox type which falls under the person domain, thereby causing over coverage to become incomplete.

There are a few different ways to resolve this issue, to varying degrees of satisfaction. We could only perform vertical search across domains where it is clear what infobox types fall into that domain, or we could accept this lack of coverage, where some infobox types might be missing but we would at least have most types or the most important types. The third option is to try to build an ontology of infobox types, where the relationship of different entities is shown. Then, we would be able to see, for instance, that "NBA player" and "NFLretired" are subentities of "Person", and every time the ontology is updated we would also update our list of which infobox types represent people. If it is to build such an ontology, or if we have access to such an ontology, then this is likely to be the best solution, but then we must consider whether the database used can efficiently integrate such information. For example, a database in which each entry explicitly represents an object and which has built-in support for object relationships would be more suitable than a database in which such relationship information must be manually checked in each query.

**Issue 3: Inconsistent infobox attributes across pages.** In a general search engine, each page can be treated much the same, and even if varied and sophisticated metadata is associated with each page, the attributes that each page can have remains within a relatively small set. When we seek to take advantage of the infobox data within Wikipedia, we see the number of
infobox types that Wikipedia’s pages contain is very large, and that each infobox type has its own distinct attributes. Each page will have values for only a very small subset of the total attributes. We might represent this relationship as a sparse matrix, in which the total attributes represent one side of the matrix and the pages represent the other. If we are using a database/schema combination in which this matrix cannot be compressed, the amount of space which must be allocated is impractical. It may be the case that a database/schema combination in which the space used is practical runs into problems with performance, whereas the schema which allows for fast performance runs into problems with space. Furthermore, there is nothing that says a new page cannot be added or a current page modified such that a new attribute is introduced. Therefore we see that a database or schema that assumes fixed attributes for each entry will have greater difficulty than one in which entries have flexible attributes.

Even if we only store a very small subset of the infobox types in Wikipedia, that is no guarantee that the number of attributes will be manageable. For example, a sample of 4 infobox types spanned a total of 2609 attributes (Table 3.1). This is due to inconsistency in attributes within pages of the same infobox type, and also introduces non-performance related problems with effective searching. One page may have an attribute called knownfor, while another page may have an attribute called bestknownfor. The data stored is similar, even identical, but if we wish to query for it, we must check for both attributes, in addition to any other attributes that the data may happen to be stored in, which we may not be aware of. In addition, there is no enforced standard for infobox format. Therefore, a country page will almost certainly contain attributes for population, gross domestic product, and other common country-related attributes, but it may also contain all manner of other attributes, which may have no or little overlap with the attributes of

<table>
<thead>
<tr>
<th>Infobox Type</th>
<th>Number of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>1094</td>
</tr>
<tr>
<td>Book</td>
<td>485</td>
</tr>
<tr>
<td>Musical Artist</td>
<td>1177</td>
</tr>
<tr>
<td>Album</td>
<td>668</td>
</tr>
<tr>
<td>Total</td>
<td>2609</td>
</tr>
</tbody>
</table>

Table 3.1: Infobox Attribute Counts
other country pages. Such attributes are difficult to make use of in queries since they apply to so few pages. As a hypothetical example, if there are a few countries with infobox attributes for annual banana production, it does not make sense to rank countries on that attribute since the winner is unlikely to actually have the highest banana production; it is simply the winner of the countries which possess that attribute. In contrast, since almost every country has an attribute for population, a vertical search engine can credibly rank countries by population.

**Issue 4: Inconsistent attribute types within an attribute.** In Wikipedia, there is no concept of types such as integer, string, floating point number, etc. for the values of attributes. Instead, each attribute value is simply text (although that text can represent a reference to another Wikipedia page or even an image). Nor are there any restrictions as to what kind of text may be used: for example, we would generally consider the page count of a book as a number, and expect the value for that infobox attribute to be convertible to a number. Instead, values for the pages attribute of book pages are in a variety of formats (Table 3.2).

Out of 14301 values, only about 33% were in the expected digit-only format. Approximately half the values had the number followed by pp, e.g. 402 pp for 402 pages. Of the remaining 15%, half were split between the alternative suffixes pages, pp., and p.. A little below 2% were lists of page counts, corresponding perhaps to different editions or formats of a book. Finally, a little over 6% of values could not be placed in any of the above categories, and include values such as ”3 vols.” and ”272-287”.

The result of this is that the values contained each infobox are not necessarily in the optimal format for searching, nor as they necessarily easy to

<table>
<thead>
<tr>
<th>Format</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>”# pp”</td>
<td>7442</td>
</tr>
<tr>
<td>”# p.”</td>
<td>362</td>
</tr>
<tr>
<td>”# pp.”</td>
<td>335</td>
</tr>
<tr>
<td>”# pages”</td>
<td>372</td>
</tr>
<tr>
<td>#</td>
<td>4655</td>
</tr>
<tr>
<td>List</td>
<td>237</td>
</tr>
<tr>
<td>Others</td>
<td>898</td>
</tr>
<tr>
<td>Total</td>
<td>14301</td>
</tr>
</tbody>
</table>
clean. The pages attribute is relatively straightforward in that it should simply be a number, yet an approach like extracting all digits from the value and storing the result would result in page counts like "3" and "272287" which are wildly inaccurate. To normalize, say, the different representations of dates and date ranges, or different ways to refer to the same person, can be a daunting task.

In addition to interpretation and cleaning of these values, we must also think about how we can store them in a database so that they can be easily queried without losing information. In many databases, a field must have a certain type associated with it, such as string, number, and so on. We could store all these page values as text, but then we would be restricted in the types of queries we could perform over this field - we could not, for example, do range queries, or query for all books of a minimum page count. If this seems like a useless query, consider that a similar problem is associated with any kind of numerical data. On the other hand, we can choose to convert the values to numbers and store them in that format. Such an approach would certainly work with the digit-only values, and many other values could be cleaned to fit this format, but not all of the data can be so easily cleaned. We would then have to choose between storing bad data (by converting "3 vols." as "3" pages and "272-287" as "272287" pages) or dropping the information because it does not fit our field type.

Furthermore, depending on the database and the chosen schema, it is not necessarily the case that we can give each infobox attribute its own field. For example, it may prove necessary, due to space constraints, to store all infobox attributes in the same field, with another field indicating the key it is associated with. In that case, if we set the field as being numerical in type, we lose all non-numerical data, whereas if we set it as being of a text type, then we lose the ability to make numerical queries or other numerical operations such as sum, average, etc.

Given such inconsistency of data, in which there is a mixture of data types, the best option would be a database in which both types of data could fit in the same field. Numerical data would be stored as integers, floats, or whatever specific type is appropriate, while non-numerical data would be stored as text. Then queries would operate over whatever values were appropriate for the query, so that a query for values greater than a certain number would not return strings because they happened to be greater in
In this chapter, we have covered various problems involved in performing vertical search over Wikipedia, both in terms of technical problems in effectively storing and querying domain-specific information and also in terms of data quality or consistency which can restrict the power or accuracy of vertical search based off such data. It is our assertion that the latter is not sufficiently serious to prevent practical and useful applications of vertical search over Wikipedia, but this is an assertion that cannot be proven or refuted in a thesis and must be tested through user experience. For now, we will consider what kind of database to use and how to organize Wikipedia’s data in order to best meet technical difficulties.
The selection of an appropriate database for our implementation is of great importance, as while it is possible to store almost any form of data in any modern database, performance and ease of use may vary considerably. The technical issues identified in the previous chapter suggest criteria to use: speed, native support for object relationships, possession of a flexible schema, and flexible attribute data types. There are also some additional features that we consider desirable. One is that the schema of the database fits well with our conception of the structure of Wikipedia and vertical search, which is that of a set of entities, each of which has a set of easily accessible attributes, including a set of links to other pages. We did not wish to pursue a framework in which the data is stored in a way that is contrary to this conception. The other desired feature was that the database be easily available for public use, and ideally open source. We did not wish to use a proprietary database or any other setup which cannot be easily imitated by those who wish to try their own hand at implementing similar vertical search.

As a result, we chose MongoDB (www.mongodb.org), which fits almost all of these considerations. MongoDB is a document-oriented database, which stores collections of documents, each of which consists of a set of keys and values. As a result, it inherits many of the features of key-value stores, such as speed and scalability, which makes it suitable for querying over a large corpus such as Wikipedia. Unlike a traditional relational database, in which each table has a fixed schema dictating the number of attributes and the data type of each attribute, MongoDB’s collections are schemaless, and each document may have any set of attributes, where each attribute, in turn, does not possess any fixed data type. Therefore, we do not have to make any special consideration for Wikipedia infobox attributes being of mixed type (number, text, date etc.), as the same attribute across many different documents may contain multiple types of data, and this attribute can be queried without any unexpected behavior (e.g. text appearing as a result for
a query for \( x \leq 50 \).

MongoDB also satisfies our additional considerations of availability and conceptual similarity. It is open-source, and can be freely downloaded from the Web. The manner in which it stores data as documents with various key-value attributes closely parallels our conception of Wikipedia storing data as pages containing various attributes. Thus, we can simply make a one-to-one mapping between pages and documents, and this is in fact the manner in which we have chosen to organize our data within MongoDB.

We also chose to implement vertical search over an alternative database, both to determine its feasibility and also in order to test our assumptions concerning desirable features of MongoDB. Since MongoDB is not a relational database, and most databases follow the relational model, a relational database was the obvious alternative. Of these relational databases, we chose to use SQLite, which is simple to use and has some of the same desirable properties as MongoDB. SQLite is fast, widely used, and open-source, and unlike most other relational databases, it is tolerant of Wikipedia’s inconsistent data types as it does not assign fixed types to each attribute or column. Rather, types are assigned to each individual instance of the attribute, and each attribute has an affinity to a certain data type, to which it will attempt to convert its values. As an example, if an attribute has integer affinity, then it will attempt to store values for that attribute as integers, but if this is not possible, it will store them as strings instead of declaring the values to be invalid.

Since SQLite does not possess a dynamic schema, we cannot adopt a similar setup to MongoDB’s in which we store all the pages in a single table, with each tuple corresponding to one page or entity and each column corresponding to one attribute, as the vast majority of elements for each tuple would be empty and the number of columns would be both enormous and subject to change as new pages were added. Instead, we adopted a setup more typical for relational databases, in which each page is split across multiple tables. Data which is common across all pages, such as title and template (infobox) type is stored in one table, while data which varies across pages, such as infobox type, sets of links, etc. is stored in other tables, with a column indicating which page the data is for. Since we worked with a small set of Wikipedia data, we only store these 5 attributes (page title, infobox type, infobox attributes, links), although further information such as page id,
text content, categories, etc. can be readily incorporated by extending the schema. As previously mentioned, one of our criteria for database selection was whether the schema necessary for data storage corresponded well to our intuitive conception of Wikipedia, and in this regard we found MongoDB’s 1-to-1 entity to document mapping to be much preferable to the mapping we were forced to adopt for SQLite, in which each page is mapped to a large number of rows spanning multiple tables. We also felt that the added complexity would result, one way or another, in slower performance.

Neither MongoDB nor SQLite features native support for object relations, and we chose to avoid such databases due to a lack of prior experience. In addition, time constraints prevented us from incorporating infobox relationship data, and without such data we would not have been able to take advantage of such a feature. For now, we have chosen to concentrate on ensuring that basic search is efficient and feasible.
Over the course of our work, we experimented with multiple approaches for retrieving information from Wikipedia and populating our databases. Initially, we chose to obtain dumps of the English content Wikipedia from Wikimedia (download.wikimedia.org) and applied a preprocessor written by Evgeniy Gabrilovich [8] [9] [10]. We then used a script of our own to parse the XML output of the preprocessor and populate our databases. However, we found that our implementations of vertical search did not need to access the full corpus or the full content of each page, and it become more convenient to make use of the dumps available on DBpedia (wiki.dbpedia.org). DBpedia provides dumps of various page content from Wikipedia such as titles, short and long abstracts, infobox properties, links, and so on, and by using these dumps it was easier to work with a chosen subset of Wikipedia. The use of these DBpedia dumps results in similar data as that obtained through directly handling the Wikimedia dumps, except that DBpedia subjects its datasets to some degree of cleaning. However, in our database designs we do not assume that our input data is cleaned or processed, and the presence of such datasets as input is not strictly necessary. Overall, for our purposes of showing vertical search over Wikipedia, we used a set of 174,739 articles, and stored the titles, infobox attributes, and links for each page.

The presence of a dynamic schema in MongoDB and a fixed schema in SQLite, along with differences in how they store data, caused us to organize data differently between the two databases. In MongoDB, the way we stored data was very similar to our previous conception of vertical search. We used 1 collection, consisting of documents which each represented a single Wikipedia page. Each document in turn consisted of a set of key-value pairs, which were the infobox attributes, in addition to the infobox type, title, and a list of links, which were stored as a single attribute. In SQLite, we used one table to store the list of pages in the form of 2-tuples of title and infobox type. Each category of attribute then occupied its own table, so we stored
the links in one table as 2-tuples of title and link and the infobox attributes in a third table as 3-tuples of title, attribute name, and attribute value. The primary difference between these two schemas is that the MongoDB schema followed our conceptual model of vertical search, with a single set of entities, each of which in turn held a set of attributes. The SQLite schema, on the other hand, was quite different. It consisted of three sets, each of which held different types of information: one held the entities and an attribute (infobox type), the second held the entity infobox attributes, and the third held the links for each entity. The different infobox attributes for each entity were held in different rows of their table, and the same was true of the links for each entity. Therefore, when we retrieve an entity from the database, if we want more than just its title and whatever attribute we queried on to obtain it, we must assemble the attribute by combining the different tuples that hold its data.

We tested the functionality of both databases by creating concrete implementations of the three example vertical searches of chapter 2, using about 100-150 lines of Python code each. By doing this, we were able to demonstrate functional vertical search, and also learn the severity of some of the difficulties in making use of Wikipedia data. We also saw how the same vertical search must be implemented differently across both databases, due to the differing schemas, and how the ease of implementing vertical search is a result of how closely the database schema matches the conceptual model. For example, in the book domain search, we query for books on their various attributes such as author, title, etc. - let's say we want to find the book "War and Peace". With MongoDB, the code is very simple:

```python
q = {'template_type':'infobox book', 'title':'War and Peace'}
results = collection.find(q)
```

The query simply specifies that the document must be a book entity where the title is "War and Peace". The results variable then consists of that specific book and all of its attributes. Querying over SQLite is much more involved because of the need to fetch data from multiple tables together, followed by combining this data into one object to represent the entity.

```sql
q = "SELECT pages.title, template.key, template.value
FROM pages, template
```
WHERE pages.template_type = 'infobox book'
AND pages.title = 'War and Peace'
AND template.title = pages.title
UNION
SELECT pages.title, 'link', links.link
FROM pages, links
WHERE pages.template_type = 'infobox book'
AND pages.title = 'War and Peace'
AND pages.title = links.title"

cursor.execute(q)
results = dict()

for (title, attribute, value) in cursor:
    page = results.get(title)
    if page is None:
        results[title] = dict()
        page = results[title]

    pagevalue = page.get(attribute)
    if pagevalue is None:
        page[attribute] = value
    elif type(pagevalue) == type(list()):
        page[attribute].append(value)
    else:
        page[attribute] = [pagevalue]
        page[attribute].append(value)

The first part of the code involves gathering together all the information for the book. First, we search in the pages table for all rows which represent books, and then within those rows for any books called "War and Peace". Once we find this book, it must be joined with the infobox attributes table ("template"). We repeat this process in order to obtain all the link information, and if we stored other data, such as text or categories, this step would grow proportionally longer (although some optimization could be performed by only performing the search for the book title once). Since the infobox
data and the links data have a different number of columns, we create an extra column for the links result so that the two results can be combined, and then take the union. This gives us a list of tuples, of the form (title, entity attribute, entity value).

The second part then consists of combining all of these tuples together into one object. Because sometimes we have multiple values for the same attribute, such as a page having a multiple links, we must add additional code which creates and appends to lists when necessary. Finally, we end up with the same result as the MongoDB query. It is self-evident which block of code is preferable, especially when we consider that the SQLite query must be written carefully in order to make sure that the database’s indexes are used. Failure to do this can result in behavior such as two complete tables being joined together, which is extremely time-consuming.

In addition to differences in ease of implementation, we also confirmed that the problems raised in chapter 3 are significant. In the book domain search, we saw that Wikipedia’s inconsistency in data presents a significant hurdle. We have already covered the distribution of different formats for the "pages" attribute of book pages in Wikipedia, but other attributes are subject to much the same issues. ISBNs can be found in different formats, such as "0-201-00650-2" and "0091073901", and some books do not have ISBN data stored at all, which can be misleading if a person queries on that book’s ISBN and takes a lack of results as an indicator that either the ISBN is invalid or data for that book is unavailable in Wikipedia. While ISBNs can be cleaned and queried over in a simple manner, the task of normalizing date values such as "1973-01-23 00:00:00", "1987", "December 1995", and "Originally 1994, then JKP in 1998" into one format is not so simple a task (although admittedly, the last value is an uncommon case).

We also found that a page can be mapped to many different kinds of entities, depending on the level of generality desired, but currently, we can only map each page to the most specific entity. In the person domain search, we only use those pages where the infobox type is "person", but there are many other infobox types which refer to subsets of people, such as NBA players, football players, etc. - we do not currently have any way to recognize that these entity types fall within the person entity type. As a result, many famous people, who one would expect to be covered in such a search engine, do not show up in the results because their infobox type is a reference to some
specific type of person. This indicates the need to either discover such infobox

With both MongoDB and SQLite, we were able to achieve identical search

As expected, the performance of MongoDB was superior to that of SQLite

Furthermore, the numbers above do not tell the whole story as for each

In all cases, the first query takes the longest to execute, due to OS-level
caching of recently accessed database data, but the difference is drastic for
SQLite. We speculate that this is yet another facet of the same schema
difference previously discussed. In MongoDB, since each infobox attribute
is also its own document attribute or column in the database, we can index
specifically on attributes. We cannot adopt this same approach in SQLite,
as in a relational database the table schema is fixed. Therefore, each infobox
Table 5.1: Search Performance (in seconds)

<table>
<thead>
<tr>
<th>Search</th>
<th>MongoDB</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Book</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author: &quot;Tolstoy&quot;</td>
<td>0.23</td>
<td>0.42</td>
</tr>
<tr>
<td>Author: &quot;Zelazny&quot; and Date &gt; 1980</td>
<td>0.05</td>
<td>0.76</td>
</tr>
<tr>
<td>Title: &quot;Grapes of Wrath&quot;</td>
<td>0.24</td>
<td>0.03</td>
</tr>
<tr>
<td><strong>Musical Artist</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Title: &quot;Metallica&quot;</td>
<td>0.56</td>
<td>13.67</td>
</tr>
<tr>
<td>Author: &quot;The Beatles&quot;</td>
<td>0.10</td>
<td>13.62</td>
</tr>
<tr>
<td><strong>Person</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Warren Buffett&quot;, depth 1</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>&quot;Warren Buffett&quot;, depth 3</td>
<td>0.05</td>
<td>0.12</td>
</tr>
<tr>
<td>&quot;Warren Buffett&quot;, depth 7</td>
<td>1.80</td>
<td>2.70</td>
</tr>
<tr>
<td>&quot;Bill Gates&quot;, depth 1</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>&quot;Bill Gates&quot;, depth 3</td>
<td>0.12</td>
<td>0.28</td>
</tr>
<tr>
<td>&quot;Bill Gates&quot;, depth 7</td>
<td>3.08</td>
<td>4.18</td>
</tr>
</tbody>
</table>

Table 5.2: Initial Query Performance (in seconds)

<table>
<thead>
<tr>
<th>Database</th>
<th>Search</th>
<th>Query 1</th>
<th>Query 2</th>
<th>Query 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>&quot;Warren Buffett&quot;, depth 7</td>
<td>40.05</td>
<td>2.74</td>
<td>2.70</td>
</tr>
<tr>
<td>SQLite</td>
<td>Author: &quot;Tolstoy&quot;</td>
<td>166.98</td>
<td>25.66</td>
<td>0.41</td>
</tr>
<tr>
<td>SQLite</td>
<td>Title: &quot;Metallica&quot;</td>
<td>13.06</td>
<td>13.11</td>
<td>13.45</td>
</tr>
<tr>
<td>MongoDB</td>
<td>&quot;Warren Buffett&quot;, depth 7</td>
<td>1.86</td>
<td>1.79</td>
<td>1.78</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Author: &quot;Tolstoy&quot;</td>
<td>0.40</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Title: &quot;Metallica&quot;</td>
<td>6.67</td>
<td>0.56</td>
<td>0.56</td>
</tr>
</tbody>
</table>
attribute must be stored in the same two "key" and "value" columns, and
instead of being able to index on each attribute separately, we must index
on these two all-encompassing columns. These indexes are necessary for
any kind of reasonable performance, but are much larger in size compared
to MongoDB’s. The speculation is that these indexes are too large in size
to fit into memory, and so must be loaded from disk upon the first query,
thereby accounting for the large delay. Successive queries do not exhibit
such performance issues as the relevant portion of the index already resides
in memory. It is likely that this problem increases in scope as the number of
pages in the database increases.

A related issue is that the creation of correct indexes that were tailored
to the queries of each vertical search example was very important for per-
formance, and it was not sufficient to adopt more simplistic schemes such as
individually indexing each column used by queries. The ordering of selections
in queries in order to make good use of SQLite’s indexes was equally impor-
tant, and writing the query inaccurately would result in a large increase in
execution time. In contrast, for MongoDB we were able to enjoy fast query
executions even before creating any indexes. Therefore, in practice we found
development using MongoDB to be far more attractive and less arduous than
the same development on SQLite.

To summarize, our concrete implementations confirm the issues which we
raised in Chapter 3. We can go through them one by one, comparing the
relative performance of MongoDB and SQLite for each issue.

**Issue 1: Large corpus size.** Speed was never much of an issue for
MongoDB, but SQLite had poor performance for the musical artists search.
The reason for this is not because SQLite is intrinsically slow, but rather
because of the queries which were carried out during the search. SQLite had
to piece together entities by looking up their attributes, and as a result had
to do much more work with a corresponding difference in response time. If
we were to search over the full Wikipedia corpus, instead of a small set of
infobox types as we do now, this issue increases in importance.

**Issue 2: Entity resolution.** Neither MongoDB or SQLite is specialized
for the task of holding entity relationships, but our implementations did not
attempt to discover such relationships in the first place. We found that this
problem was significant, but not insurmountable - for example, we could have
invested manual labor into discovering infobox types which are subsets of
"person", and there are ontologies available which contain this information.

**Issue 3: Inconsistent infobox attributes across pages.** MongoDB, which possesses a flexible schema, does not encounter difficulty here, but SQLite suffered greatly as its fixed schema forced us to adopt a table layout which scattered each page’s data across multiple tables, making it necessary to piece each page back together again during the search. However, even more severe is the fact that the table layout stored all infobox attributes and values within the same two columns, forcing an all-or-nothing choice between indexing the infoboxes or not. Since the searches were unusable without indexes, the former option had to be chosen, which resulted in massive indexes that did not fit in memory.

**Issue 4: Inconsistent attribute types within an attribute.** We found that both MongoDB and SQLite handled storing and querying inconsistent data well. They were able to store values of mixed types within the same column or attribute, and search over the appropriate values depending on the content of the query. However, being able to direct a query against a set of values without crashing and being able to actually match the correct values are two different matters. The variety of formats that an attribute can come in means either a corresponding number of queries must be used, in order to catch all the forms that the attribute can take, or it may simply mean that the engine is limited on which pages it can effectively search over for that attribute. This, however, is a matter of data quality and consistency, and is not a database issue.

Of these issues, the third caused the greatest rift between MongoDB and SQLite. It caused SQLite’s performance to fall behind significantly, especially on the first few queries for each domain, where the super-indexes that must be maintained are loaded into memory. Furthermore, the organization of data in SQLite, in which a page’s data is split across multiple tables, is less intuitive and harder to handle than the organization of data in MongoDB, in which a page’s data is kept in one document. As a result, even if SQLite’s performance could be improved, there does not seem to exist any motivation for attempting to do this, as it is both easier and faster to implement these vertical searches on MongoDB.

Overall, while our fears were proven valid, we were also able to prove that useful vertical search can be carried out over Wikipedia, and that it is worthwhile to further time and resources into this subject. Of the two databases,
we discovered that MongoDB performed significantly better, and so future development should take place on either this database or a database which supports object relations (the one feature which was missing in MongoDB).
Prior to implementing vertical search over Wikipedia, we were influenced by existing engines. WikiXMLDB loads Wikipedia data into a XML database, Sedna, and provides various query operations over it, such as fetching the various parts that make up an article, querying on article attributes, and fetching neighboring pages [11]. Since the dumps of Wikipedia that we were originally working with were in XML format, it seemed intuitive to also use a XML database. However, we ultimately felt that our conception of vertical search did not match well with XML’s tree format. We also found it difficult to work with navigational queries, and found the nonprocedural queries of MongoDB and SQLite to be much cleaner.

The Faceted Wikipedia Search uses the DBpedia datasets to provide faceted browsing over 19 Wikipedia pages, allowing users to narrow down a selection of Wikipedia pages based on their infobox attributes - for example, a set of persons may be filtered by their date of birth, origin, name, etc [?]. This is accomplished through queries executed over a proprietary search engine. The manner in which we have implemented our vertical searches is in many ways similar to that of the Faceted Wikipedia Search, utilizing infobox attributes as the primary method to select pages. However, the number of ways to query over Wikipedia is indefinite, while this engine only offers one query method. We chose to instead utilize open source databases and demonstrate how a variety of approaches can be taken to implementing vertical search.
We found two major obstacles in the face of vertical search over Wikipedia: the inconsistency of data format within different instances of an attribute, and difficulty in entity resolution. Over multiple pages of a given type, such as book, person, country, etc., the same attribute can be a number on one occasion and a string on another. Each string, in turn, may contain the same value formatted in different ways, or it may contain multiple numbers, where the descriptive text is beyond the capacities of an agent to comprehend. As a result, when we query on an attribute that holds values of mixed type, we are effectively only querying on a subset of the pages. For us to perform automatic cleaning of the data is an option, although in many cases it may transform valid data into invalid data through the removal of text required to provide context. Our entity resolution problems consisted of not being able to obtain all the pages which fit our domain due to the existence of infobox types which were subsets of the infobox type that we were looking for. A large portion of the people on Wikipedia do not possess the ”person” infobox, but rather some subset of that concept, such as ”NBA player”. We can easily find all pages with the ”person” infobox, but to find these sub-infoboxes is more difficult.

Various methods have been attempted to rectify this and bring greater order and consistency to Wikipedia infobox data. One possibility is to compile an ontology of various infobox types / objects, their relationships to one another, the attributes of each infobox, and what type they are to be. Various heuristics and techniques can be used, such as examining the names of attributes for clues indicating their meaning such as the presence of the word ”birth”, or the use of a name parser to normalize names [12]. The DBpedia ontology is compiled by hand, resulting in a hierarchy of some 259 classes with a total of 1200 properties and fixing some inconsistencies in Wikipedia’s infoboxes such as the presence of multiple infoboxes for the same concept [13]. However, DBpedia’s manpower is not sufficient to cover all infoboxes
on Wikipedia, nor is it likely to match the manpower of those users whose contributions add new inconsistencies to Wikipedia on a daily basis. It may ultimately be necessary for this consistency of data to exist within Wikipedia itself, through the enforcing of standards or user-driven cleanup of existing pages, rather than being created in some external dataset. Nevertheless, in the short term, the incorporation of such a cleaned dataset and ontology appears to be the best and most obvious choice for future improvements.
The rise of vertical search engines on the Web leads to the question of whether it is feasible to adopt the same approach to searching across other, similar document collections. In this thesis, we argue that the answer is yes, and put forth examples of how to conduct a vertical search as well as anticipating potential problems and their solutions. We then put our theories to the test by implementing practical examples of vertical search across a subset of Wikipedia data using two databases. These implementations show that vertical search across Wikipedia is indeed viable, but that problems we identified are also significant, and must be handled with care. Further work in this area seems promising for more fully resolving these problems.
REFERENCES


