Bulk Loading Large Collections of Hyperlinked Resources

Davood Rafiei
Department of Computing Science
University of Alberta
drafiei@cs.alberta.ca

ABSTRACT

The problem of loading large collections of hyperlinked resources into a relational database is complicated with internode references when these references cannot be indexed. We show that this scenario can arise in many real life hyperlinked resources and propose several solutions to address the problem. We run some experiments over a graph of the Web with 178 million nodes and around 1 billion edges and report our results.

Categories and Subject Descriptors: H.3 [Information Storage and Retrieval].

General Terms: Algorithms, Experimentation.

Keywords: bulk loading, larges graphs, Web graph.

1. INTRODUCTION

Useful information can be extracted and explored from the linkage structure of the hyperlinked resources, leading to better understanding of the topology of the network. Storing data in a relational database has the benefit that queries can be easily written in SQL and optimized within the SQL engine. Loading a small graph into a database is straightforward. Loading a large graph when there are no *unresolved references* is also trivial. Loading a large graph in the presence of *unresolved references* is not only straightforward, but it is often the source of a mis-understanding. For instance, Bar-Yossef and Rajagopalan [2] use tables `PAGES(page_key, page_shingle)` and `LINKS(src_page_key, dest_page_key)` to store a Web graph and make the claim that building these tables "requires a constant time per page." It is easy to show that this claim does not hold for the Web graph unless more specific assumptions are made. First, the node descriptions must be shortened because a URL is too long to be used as a key. While a URL on average is 53 characters long, machine-generated URLs can be longer than 1000 (based on our experiments with a snapshot of the Web). This is far more than the maximum length allowed for candidate keys or indexable columns in major database systems. The length of both a primary key and an indexable column is limited, for example, in DB2 to 254 characters and in Oracle to 758 characters. Apart from DBMS limitations, long keys incur unnecessary additional costs at the query processing time. A solution is to assign a shorter id to every URL. Second, when a node is assigned a new id, all references of the node must be replaced with this id. Resolving references in general cannot be done in linear time.

The aforementioned problems are not unique to the Web graph. Machine-generated ids (e.g. URIs) can appear in xml documents through the use of xlink and name spaces. Long ids can appear in academic citations to refer to, for example, publications inside digital libraries. Long ids can arise in networks with no centralized authority governing the creation of ids such as peer-to-peer networks.

If a referenced node is already in the database (those are called *backward references*), the id assigned to that node may need to be looked up. If a referenced node is not yet loaded (those are called *forward references*), the references may be saved until the nodes they refer to are loaded. Our work addresses the general problem of loading a graph when neither the graph nor the *id map* can fit in memory.

In this paper, we evaluate some of the alternatives for loading a large graph into a relational database; our evaluation includes the expected behaviours and some of the limitations. We present the conditions under which the loading can be done in linear time, and an algorithm which can be used when the conditions are met. As a proof of concept, we report our experiments of loading a connectivity graph of a snapshot of the Web into DB2.

Related to our work is the problem of bulk loading into an OODB database where some solutions have already been proposed. Some of these solutions load the entire list of references into memory, thus these methods are not applicable to large graphs. Only a partition-list approach (which corresponds to our join-based resolution) does not make such an assumption and can be applied to large graphs [11].

2. LOAD ALGORITHMS

Let *Nodes* and *Edges* denote the two tables which will store the graph description. Each node is uniquely identified by one or more attributes (e.g. URL, ip address, etc). The load algorithm (1) must assign a short key to each node and (2) must replace all references to that node with the newly-assigned key. In our study, we examine four different load algorithms. Our *naive* algorithm is basic and is only used for the purpose of comparison. Our work with Web-like graphs that have certain power law properties raises

HT'05, September 6–9, 2005, Salzburg, Austria.

some interesting questions such as: (1) can we exploit those properties to boost the performance of the loading? (2) how much improvement can be achieved? Our *semi-naive* algorithm improves upon our naive algorithm and addresses some of these questions. For clarity of the presentation, both algorithms are geared toward the Web graph. Our third and fourth algorithms are general and perform the loading in two steps: first the references are resolved, then the data is bulk loaded using a relational bulk loading utility.

2.1 Naive Loading

```
Stream through Web pages and for every page r
  If r has not been seen before, then add (rid, r)
  to nodes where rid is an id assigned to r
  For every link s inside the Web page
    If s has not been seen before, then add
    (sid, s) to nodes where sid is an id assigned to s
    Add (rid,sid) to edges
```

The algorithm performs a lookup for every URL that is encountered; this includes lookups for the URL of the page and for the URL of every page the page links to. For the Web graph, it is estimated that a page links to 7.2 pages on average [8]. Thus the average number of lookups for a document is estimated to be 8.2. The algorithm does an insert for every node and every edge.

Column URL is too long to be indexed (as discussed earlier). On the other hand, without an index on URL, each lookup from *nodes* will end up scanning the whole table. To avoid a sequential scan, we break column URL into two pieces, say url_1 and url_2, such that url_1 is short enough to be indexed and $URL = concat(url_1, url_2)$. Having an index on url_1, we can benefit from the index for lookups.

The algorithm still has some drawbacks including the large number of lookups, the cost associated to each lookup and the number and the cost of inserts to the index on url_1. There is also an overhead due to the logging activities of the DBMS, but it is possible to avoid such an overhead by building data files outside the DBMS and loading them into the DBMS afterwords using a relational bulk loading utility.

2.2 Semi-naive Loading

The naive algorithm can be improved in several aspects. Our improvements are geared toward graphs with scaling properties. In particular, we exploit the power law distributions of in- and out-degrees of nodes to improve the load performance. It is shown that a large number of dynamic systems demonstrate such power law distributions [1]. Since each incoming edge to a node translates to an access of that node in the naive algorithm, clearly nodes with the largest in-degrees are accessed the most. As our first improvement of the naive algorithm, we identify a set of nodes with the largest in-degrees and store them in a hash table in memory. No disk access is needed for searches over nodes that reside in memory. It is easy to compile such a list from a relatively small sample; if a url appears in a large number of pages, it is likely that it will also appear in a small sample [5].

The Web graph also has a large number of nodes with zero or one incoming edges. When every such node is visited, there is no need to search the database since the node cannot be there. But in the naive method, those nodes are still being searched before being loaded into the database. To avoid unnecessary searches, we build a summary of all URLs in the database and use it for membership testing. This is done using the Bloom filters [7], a hash-based trick for quickly testing membership. The summary is built incrementally while the nodes are being loaded.

Our third refinement is specific to the Web where links can appear more than once in a page. On average, more than 14% of the links in a page are duplicates, based on our experiments. To avoid searches for duplicate links, we keep in a memory hash table all links recently visited in the page.

2.3 Bulk Loading Using Joins

Assign a unique key to each node, and use joins to replace every node description in edges with its surrogate key. The join is done twice so that both endpoints of an edge are replaced with their keys. For a shorter key assignment, one can use a dense assignment of sequential ids (e.g. integers). The node set may include duplicates in which case duplicates are removed before keys are assigned. If ρ_e and ρ_n respectively denote the number of disk pages in *Edges* and *Nodes*, the cost of this method is mainly the cost of joins which can vary from $3(\rho_e + \rho_n)$ I/Os (when the main memory is large enough to store $\sqrt{\rho_n}$ disk pages) to up to $O(\rho_e log\rho_e + \rho_n log\rho_n)$ I/Os in general (e.g. see [4]).

2.4 Bulk Loading with Fingerprints

Map node descriptions to short ids using a fingerprinting function. If it can be guaranteed that every node description is mapped to a unique id, then all references can be resolved within one scan of the nodes and the edges. It is not hard to show that there is no such mapping that works for all possible inputs, though there are families of functions such that, for all possible inputs, functions in these families result into a very limited number of collisions. One such family of functions is discussed here.

Next, find every node that is assigned a non-unique id and replace it with a unique key. This can be generally done by sorting the nodes and reassigning some of the keys; the cost in terms of the number of I/Os varies from $3\rho_n$ (when the memory can hold $\sqrt{\rho_n}$ disk pages) to $O(\rho_n log\rho_n)$. Denote with N_{ra} the set of nodes where a key is reassigned.

Finally scan the edges and replace every node description with its id. This can be easily done using the same function applied to nodes except for nodes in N_{ra}. If the number of nodes in N_{ra} is small, then those nodes can be kept in memory and can be looked up before an id is assigned.

In our implementation of this method, we had to choose a fingerprinting function that could avoid duplicate ids as much as possible since these ids incurred additional costs. We decided to use Rabin's method [9] which has also been used in several practical applications (e.g. [6]). Compared to alternative fingerprinting functions such as MD5 and SHA [10, 3], Rabin's method has the flexibility of generating fingerprints of any length, hence it is applicable for graphs of different sizes. The length of a fingerprint in MD5 is fixed to 128 bits and in SHA is fixed to 160 bits or longer. These other methods also do not provide a bound on the expected number of collisions.

Rabin's Fingerprinting Method

Given an input sequence $b_{m-1}, \ldots b_1, b_0$ of m bits, the sequence can be regarded as the polynomial $P(x) = x^m + b_{m-1}x^{m-1} + \ldots + b_1 x + b_0$. Let $Q(x) = x^k + c_{k-1}x^{k-1} + \ldots + c_1 x + c_0$ be an irreducible polynomial [1] of degree k.

[1] A polynomial is called irreducible if it cannot be factored

The reminder $P(x)\ mod\ Q(x) = h_{k-1}x^{k-1} + \ldots, h_1 x + h_0$ is computed using polynomial arithmetic modulo 2; the sequence $h_{k-1}, \ldots, h_1, h_0$ is the fingerprint of the input sequence. This method has a number of interesting properties (as shown by Rabin [9]).

LEMMA 1. *For an irreducible polynomial $Q(x)$ of a prime degree k, this algorithm assigns every fingerprint with equal probabilities.*

LEMMA 2. *Given n sequences, each of length m bits or less, and a prime number k that denotes the length of the fingerprints in bits, the probability that two distinct sequences are mapped to the same fingerprint is less than $nm^2/2^k$.*

The ratio in Lemma 2 is an upper bound and can be greater than 1. Lemma 2 bounds the expected number of collisions to $n^2m^2/2^k$.

3. EXPERIMENTAL RESULTS

This section reports our experiments with loading a graph of the Web with over 178 million nodes and 800 million edges, crawled in 1999 by by Internet Archive. Due to our hardware limitations, our experiments were conducted only using the crawled Web pages and the hyperlinks induced on such pages. The graph would have had over 1.3 billion nodes if we wanted to include pages that were not crawled. Our experiments were conducted using DB2 on a Linux machine with dual Pentium III processors running at 933 MHz, 2GB of RAM and a RAID disk of size 640GB. The nodes and the edges of the graph were respectively stored in tables *nodes* and *edges*.

We used the Linux commands for most of our data preprocessing. The join-based resolution was implemented by sorting the hyperlinks twice (once on the source URL and once on the destination URL) and the URLs and merging them. For the resolution by fingerprinting, we only needed to scan URLs and hyperlinks once. We relied on the DB2 bulk loading utility to identify the URLs that were assigned the same ids; this incurred no additional costs since the bulk loader was already checking for the uniqueness of the primary key and our fingerprinting scheme guaranteed to keep the number of collisions low (if not zero). The join-based resolution took over 750 minutes whereas the fingerprinting scheme only took 34 minutes. Once the references were resolved, we used the DB2 load command to bulk load data into the database; the running time of this step for both algorithms were the same. The load time did not include the checking for the referential integrity constraints as they were not needed.

For the naive and semi-naive algorithms, we could not load more than a tiny fraction of data after running them for a few days. As shown in Figure 1, the running time of each *insert* statement increases at least linearly with the load factor.

4. CONCLUSIONS

We presented some of the issues related to loading a large network into a relational database and discussed some possible solutions, two of which used the relational bulk loading utility. Both join- and fingerprinting- based algorithms are scalable to larges graphs but the fingerprinting scheme is quite faster, as reported in our experiments. There were other factors that affected the load time and were outside our control. They included the time for data cleaning and also the time for running the relational bulk loading utility.

into nontrivial polynomials over the same field.

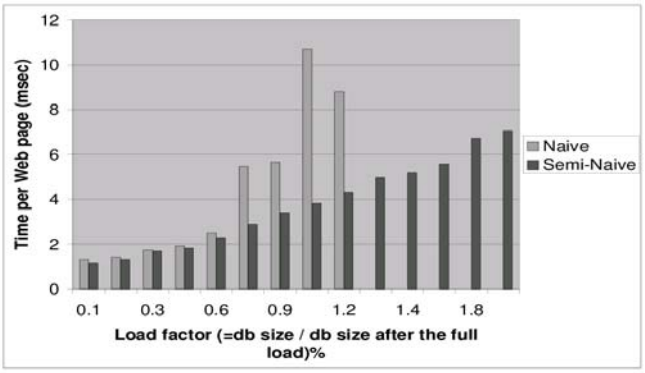

Figure 1: Time per adding a node varying the database size

Acknowledgments

This work is supported by Natural Sciences and Engineering Research Council of Canada. Rabin's method was implemented by Calvin Chan and Hahua Lu.

5. REFERENCES

[1] R. Albert and A. L. Barabasi. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–94, 2002.

[2] Z. Bar-Yossef and S. Rajagoplan. Template detection via data mining and its applications. In *Proc. of the WWW Conference*, pages 580–591, 2002.

[3] FIPS. Secure hash standard. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[5] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the Web. In *Proc. of the WWW Conference*, pages 213–225, 1999.

[6] A. Heydon and M. Najork. Mercator: a scalable, extensible web crawler. In *Proc. of the WWW Conference*, pages 219–229, 1999.

[7] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, second edition, 1998.

[8] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the Web. In *Proc. of the VLDB Conference*, pages 639–650, 1999.

[9] M. O. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Center for Research in Computing Technology, Harward University, 1981.

[10] R. Rivest. Rfc 1321 - the MD5 message-digest algorithm. http://www.faqs.org/rfcs/rfc1321.htm.

[11] J. L. Wiener and J. F. Naughton. Oodb bulk loading revisited: The partitioned-list approach. In *Proc. of the VLDB Conference*, pages 30–41, 1995.