Design Alternatives for Large-Scale Web Search: Alexander was Great, Aeneas a Pioneer, and Anakin has the Force[*]

Matthias Bender[†], Sebastian Michel[†], Peter Triantafillou[‡], Gerhard Weikum[†]

[†] Max-Planck-Institut fuer Informatik
66123 Saarbruecken, Germany
{mbender, smichel, weikum}@mpi-inf.mpg.de

[‡] RACTI and University of Patras
Rio, 26500, Greece
peter@ceid.upatras.gr

ABSTRACT

Indexing the Web and meeting the throughput, response-time, and failure-resilience requirements of a search engine requires massive storage and computational resources and a careful system design for scalability. This is exemplified by the big data centers of the leading commercial search engines. Various proposals and debates have appeared in the literature as to whether Web indexes can be implemented in a fully distributed or even peer-to-peer manner without impeding scalability, and different partitioning strategies have been worked out. In this paper, we resume this ongoing discussion by analyzing the design space for distributed Web indexing, considering the influence of partitioning strategies as well as different storage technologies including Flash-RAM. We outline and discuss the pros and cons of three fundamental alternatives, and characterize their total costs for meeting all performance and availability requirements. We give arguments in favor of a system design based on term partitioning over a DHT-based peer-to-peer network with modern top-k query processing and a judiciously designed combination of disk and Flash-RAM storage, and we show that this design has intriguing properties and a very attractive cost/performance ratio.

1. INTRODUCTION

1.1 Motivation

Indexing the Web for fast keyword search is among the most challenging applications for scalable data management. Major search engines such as Google use gigantic computing power, with large data centers consisting of thousands of low-end computers (i.e., PC technology in customized racks), very carefully designed techniques for scalability and availability (i.e., caching, redundancy, load balancing), and customized software such as GFS [4, 12]. Although detailed figures are not publicly known, it is estimated that Google indexes about 20 Bio. Web pages and needs to sustain a throughput of about 2,000 queries/s (i.e., 150-200 Mio. queries per day) with peak loads being up to 10,000 queries/s. In addition, there are stringent response time requirements: user-perceived latency must not exceed 1 s; with wide-area network transfers taking several hundred milliseconds, this requirement typically translates into a desired upper bound of 100 ms on the data-center side.

Google, Yahoo!, and MSN can obviously meet these requirements, but there is a high cost involved for operating such huge data centers. Moreover, it is not clear how much more this architectural approach can be scaled up at reasonable cost if both data and workload continue to grow in the coming years. Baeza-Yates et al. estimate that by 2010 a big search engine like Google will need one million computers [3]. And if such numbers are not perceived as a mega challenge (even literally), consider a futuristic search engine for a comprehensive Web Archive. The Internet Archive (archive.org) aims to preserve the history of the Web, but even its current 500 TB collection, containing more than 1,000 Bio. page versions, is very partial at best and allows only URL-based access in the temporal dimension. A reasonably complete Web archive with support for full-fledged time-travel querying is way beyond today's feasibilities for scalable data management.

Therefore, designing and investigating alternative architectures to the current ones is highly relevant for future scales. Distributed indexing and peer-to-peer (P2P) systems are intriguing approaches, but it is not clear if and how they can cope with the gigantic dimensions of the Web and the challenging workload of an Internet search engine. Li et al. [14] raised doubts that a wide-area distributed architecture like a P2P network would be feasible at all, but more recently various projects on P2P search (e.g., [6, 18, 7, 26, 16, 22]) and high-performance indexing (e.g., [20, 2]) have developed techniques towards scalable architectures that may possibly become practically viable. Baeza-Yates et al. [3] pose distributed indexing for Web scale as a grand challenge and outline various research routes that give hope.

1.2 Contribution

[*]This work is partially supported by the EU project AEOLUS.

ACM SIGIR Workshop on Large Scale Distributed Systems for Information Retrieval '07, Amsterdam, The Netherlands.

This paper intends to outline and analyze several architectures that may lead to scalable solutions, while in the mean time picking up the challenges put forth by [14, 3]. The main purpose of the paper is to shed more light into the overall design space and understand the pros and cons of different approaches. We consider three major dimensions for structuring the design space and in particular:

- the partitioning of the index:
  - partitioning by documents (i.e., Web pages, news or blog items, photos, etc.) versus
  - partitioning by content feature (i.e. text terms or Flickr-style tags)
- the way index partitions are distributed and handled during query processing:
  - by utilizing clusters of computers in data center environments versus
  - utilizing large-scale geographically distributed networks (such as DHTs), and
- the storage technology that is primarily used for holding index partitions on a computer:
  - RAM (the current choice by major search engines),
  - disks (a more database-style approach), or
  - Flash-RAM which is the technology used in digital cameras, MP3 players, USB sticks, and mobile phones with a strong trend towards becoming a viable alternative to disk storage [13, 21, 8].

We present three major alternatives as particularly interesting points in the design space. We have coined these approaches Alexander, Aeneas, and Anakin, for reasons that will become clear later[1]. Each of them will be analyzed in terms of total costs needed to provide sufficient space, redundancy, throughput, and acceptable response time. Alexander largely corresponds to the current solution that large search engines employ with RAM orientation and document partitioning; Aeneas is a P2P-style distributed approach with disks for low-cost storage and feature partitioning; and Anakin is a more futuristic but most promising design with Flash-RAM and term partitioning. We also discuss the critical issues of load balancing and index maintenance for each of these architectures. All architectures can be deployed in a local-area high-speed network environment like a data center, and some are also suited for wide-area networks like P2P networks; however, the different properties of LAN vs. WAN affect the architectures' abilities to scale up and perform well.

The rest of the paper is organized as follows. Section 2 briefly reviews relevant characteristics of Web-scale data, access load, and storage technologies. Section 3 presents the Alexander design. Section 4 presents the Aeneas design. Section 5 presents Anakin. Section 6 discusses load balancing issues for the presented approaches. Section 7 discusses the issues of index construction and maintenance, before Section 8 concludes this work and points at future research directions.

[1]The sequence Alexander, Aeneas, Anakin reflects history, and we are increasingly moving away from reality to fiction - mythology and modern subculture.

2. BRIEF REVIEW OF TECHNOLOGY AND WEB SCALE

We briefly review some Web properties and technology parameters that are relevant for our comparison of architectures. We assume that the Web has 20 Bio. pages, each with an average of 500 bytes indexable information (i.e., excluding embedded images etc.), which yields a total index size of 10 TB. Another way of looking at index size is in terms of the number and length of index lists in the corresponding inverted file. We assume that there is a total number of 10 Mio. distinct terms (keywords and their many variations incl. names, typos, numbers, etc.) and that each Web page contains 100 indexable terms on average. This creates 20 Bio. * 100 = 2 * 10^{12} = 2 Trio. postings to the index where a posting captures a *(document, term, score)*-triple. Obviously, many Web pages have way more than 100 terms, but a large fraction of the 20 Bio. pages is known merely through href anchors pointing to them and such anchors include only a handful of terms. So an average of 100 terms per document seems reasonable. The average length of an index list in the inverted file thus is 2 * 10^{12} / 10^7 = 2 * 10^5 postings. Each posting has a raw length of 10-20 bytes (two ids and a numeric score), but inside an index list they are highly compressed. We assume an amortized length of 5 bytes per posting. This results in a net length of 10^6 bytes = 1 MB per index list. All 10 Mio. index lists together consume 10 TB (which is consistent with our first calculation).

We assume that a standard computer used in a cluster or a P2P network is a low-end PC but configured with ample memory, say 4 GB, and costs $1,000. We consider a single 1 TB disk at a cost of $500 as a standard unit for disk storage (we will later consider also small RAIDs with coarse-grained striping for higher sequential bandwidth). The disk has a random access time of 5 ms for a 10 KB block, thus yielding up to 200 random IOs/s, and a sequential transfer rate of 20 MB/s, yielding up to 2,000 sequential IOs/s (for transfers of multiple, contiguous 10 KB blocks). Finally, for Flash-RAM we assume 16 GB units at a price of $200. Flash-RAM has about the same sequential transfer rate of 20 MB/s offered by a disk, but it has much shorter random access time (it is some kind of RAM after all), namely, 0.1 ms random access time for a small 2 KB block. This results up to 10,000 random IOs/s for small blocks and up to 2,000 sequential IOs/s for large blocks.

For query processing we assume that typical queries include only a few keywords, and for simplicity our crude analysis of query processing costs here assumes single-term queries. So for each query we need to fetch the corresponding index list from disk if it does not reside in RAM or Flash-RAM and we need to process it. We assume that the processing has a fixed startup time of 1 ms and then needs 1 ms for every 1,000 postings, that is, 200 ms for a full index list with $2 * 10^5$ postings. Reading the full list from a single disk takes 1 MB / 20 MB/s = 50 ms. Reading from disk and processing can be pipelined; so the processing time determines response time and throughput on a single computer. The sustainable throughput with a single computer is therefore 5 queries/s. Obviously, partitioning lists across multiple computers in a cluster is an effective way of scaling-up.

In the following sections we use the above parameters for crude cost-and-performance models. We will initially assume that (i) data and load are perfectly uniform, (ii) the same length for each index list and (iii) the same frequency of each term in the query workload. This is a big oversimplification, and we will revisit it in Section 6 on load balancing where we will assume heavily skewed data and load.

3. ALEXANDER: RAM-ORIENTED, DOCUMENT-PARTITIONED CLUSTERS

The first architecture we consider mimics more or less what major search engines such as Google do today. We use the name Alexander, after Alexander the Great, the great Greek strategist well known for a number of feats, including the brute-force cutting of the Gordian knot (not a bad method after all, whether real or myth). In technical terms, a good way of characterizing this architecture would be as "embarrassingly scalable"[2]. The key means to achieve great scalability are

1. hash-partitioning the complete index by document ids and spreading the resulting partitions (referred to as "shards" in Google's architecture) across a few hundred computers so that each computer has a relatively small random fraction of a list's postings,
2. keeping all these partitions for all terms in RAM, and
3. replicating such a cluster a number of times for higher throughput and availability.

If we want to keep the entire index in RAM, we need 2,500 computers for one copy of the index (recall: each computer has 4 GB memory, and the total index is 10 TB). If we want to leave some leeway for workspace memory, we can assume that we need 3,000 computers. This is one cluster. Each computer, in principle, holds 1/3,000-th of every index list (we may deviate from this simple scheme by avoiding overly small partitions and using a smarter combinatorial or randomized assignment, but this detail does not really matter for the big picture).

Every query that arrives at such a cluster will now be executed by all 3,000 computers in parallel. According to our query processing model of Section 2, this takes 1 ms + 200/3000 ms $\approx$ 1 ms and occupies all computers simultaneously. This gives us a sustained throughput of 1,000 queries/s. To satisfy the peak throughput requirement of 10,000 queries/s, we need 10 such clusters, with a total of 30,000 computers. This also provides sufficient redundancy for fault-tolerance and very high availability. The response time requirement of 100 ms is easily met, assuming that the arriving query load is evenly spread across clusters which results in low utilization and thus only short queuing delays during normal load. During peak load, with 10,000 queries/s, queuing delays would no longer be negligible, but we do not further elaborate on this and rather assume that we stay well below this "maximum" load almost always (or otherwise increase the number of clusters from 10 to say 15). The total cost of the entire system, 10 clusters with 3,000 computers each, is \$1,000 * 30,000 = \$30 Mio. A similar calculation is presented in [3] (with assumed costs of \$3,000 per computer which may include operational costs for the data center and its staff); our calculation is more detailed because we want to perform analogous analyses for other architectures, too.

[2]In the eighties, the phrase "embarrassingly parallel" was used for algorithms that could be parallelized with perfect speedup in an almost trivial manner. Database scans are one example.

A strong point of the Alexander architecture, probably its most compelling property, is the perfect load balance. Every computer that participates in executing a query performs the same share of work, by virtue of the hash partitioning on document ids. This property, in combination with the general simplicity of the architecture, allows perfect scalability. If the index size increased by a factor of 10, we would simply increase the number of computers per cluster from 3,000 to 30,000. If the index size stayed invariant but the load increased by a factor of 10, we would increase the number of clusters from 10 to 100. Of course, the cost of the overall system also increases linearly. On the downside, this embarrassing scalability works only for simple data like keyword indexing and simple queries like keyword queries. It is not clear how to make this approach work for sophisticated data like big tables with foreign-key relations or entity-relationship graphs and advanced operations like joins, proximity search, etc. (Google, Yahoo!, etc. would have efficient methods for phrase matching, proximity search, etc. as well, but certainly rely on the fact that advanced queries of this kind constitute only a tiny fraction of the overall workload.)

4. AENEAS: DISK-ORIENTED, TERM-PARTITIONED P2P NETWORK

In terms of scalability, the Alexander design cannot be beaten. But it has significant cost. Our alternative designs therefore aim to provide good scalability at much lower cost (i.e., with fewer computers). To this end, we switch from document partitioning to term partitioning, and we consider ways of replacing RAM by less expensive disk or Flash-RAM storage.

Term partitioning is intriguing because it leads to a data placement that allows serving a single query from only a few computers (one for each keyword in the query), as opposed to using thousands of computers (i.e., a whole cluster) on behalf of every query. So we intentionally reduce intra-query parallelism. This is a well-known recipe for increasing throughput or reducing cost/throughput [10, 17, 15, 24]. The same situation arises in RAID-style storage systems or in parallel database systems, and the standard technique is to allow only coarse-grained parallelism (e.g., by block or track striping as opposed to byte striping [9]. On the other hand, the challenge that we face is potential load imbalances. For now we will still assume uniform load which eliminates the danger of load imbalance by definition. Our design uses hash-partitioning for some degree of balancing, and we will see that the cost savings of term partitioning are so high that we can afford a safety margin of extra computers that would help countering any load imbalances. In addition, we will discuss some smarter algorithmic techniques later in Section 6 on load balancing.

Our first design in the above spirit is called Aeneas, after the smart hero in the Greek and Roman mythology. After the battle of Troy, Aeneas took an uncertain and risky course, but his descendants eventually founded Rome, the seed of another empire. The key algorithmic assets of our

Aeneas system architecture are its use of distributed hash tables (DHTs) (like Chord [25], CAN [23], or P-Grid [1]) and threshold algorithms for top-k queries [11]. DHTs are used to map (the index list of) a term to a DHT node. Hence, the index lists for the terms in the 10 Mio. vocabulary are spread across a certain number of computers. In contrast to standard hashing, DHTs that are based on consistent hashing can gracefully handle failing or otherwise departing computers and also newly joining ones. They also come with means for integrated replication. To save costs, each computer keeps the index lists that it is responsible for on disk rather than in memory. However, we can still utilize RAM for aggressive caching of popular lists.

With 1 TB disks and one disk per computer we would need only 10 computers to hold the entire index. But then our throughput would be too limited as one computer can sustain only 5 queries/s for single-term queries (recall that asynchronous disk I/O and list processing can be pipelined). But 200 computers of this kind could sustain 1,000 queries/s as each query is served by only one computer. Assuming that the typical query consists of 2 keywords and this involves two list-processing jobs on two different computers, we need 400 computers for the normal-load throughput. For peak load, fault-tolerance, and availability, we follow the Alexander design and replicate such a 400-computer cluster 10 times, arriving at 4,000 computers and a total cost of (\$1,000 + \$500)* 4,000 = 6 Mio. This is 5 times less than for the Alexander system.

We can do even better by employing a threshold algorithm for efficient top-k query processing. With these algorithms, postings in an index list are sorted in descending score (or "impact") order. The algorithm sequentially scans the index lists for the keywords in the query, aggregates score mass for result candidates, and incorporates a smart test for correct termination without having to read the entire lists. In our work on variants of these algorithms we have performed extensive experimentation with sizable collections (including the TREC Terabyte benchmark [5]). Typically, multikeyword queries need to scan only 10% or less of the index lists to compute the top-10 results (often the scan depth is even much smaller). Consequently, the work per query is only 1/10-th of the full-list processing, namely, 1 ms + 200/10 ms = 21 ms. This boosts the throughput per computer by a factor of approximately 10, and consequently we need only 40 computers per cluster to sustain 1,000 (2-term) queries/s. This smarter variant of the Aeneas design therefore needs a total of 400 computers for 10 clusters and costs \$ 600,000. This is 50 times less than for the Alexander system.

5. ANAKIN: FLASH-ORIENTED, TERM-PARTITIONED P2P NETWORK

Our third architecture builds on the Aeneas design, but replaces the disk storage by Flash-RAM. This opens up an opportunity for random access to index postings and we will see that it leads to a very attractive system design. We have named this architecture after the Star-Wars hero Anakin Skywalker (aka. Darth Vader, just in case someone does not know). We think this is appropriate as the main driver for Flash-RAM technology is the entertainment industry (cameras, MP3, portable video). Also, Anakin is a Jedi knight who can look a few seconds into the future, and our design speculates about Flash-RAM soon becoming a major technology for server-oriented systems as well (see also [13, 21, 8]).

If we want to keep the entire index on Flash-RAM storage and each computer can be equipped with 64 GB at a cost of an additional \$800, we need 10 TB / 64 GB $\approx$ 157 computers. Flash-RAM allows us to make fast random accesses to postings, which can be utilized by the random-access-oriented variant of the threshold top-k algorithm. Our experiments with these algorithms indicate that random accesses lead to much better pruning of result candidates and much faster termination. Typically, we need to scan only 1% of the index lists to compute the final top-10 if we have the luxury of extensive random lookups. In order to be able to have both fast sequential scans and random lookups, we configure the computers to have both disk and Flash-RAM storage. This way, we can use the high sequential bandwidth offered by the disk without putting load on Flash-RAM, and utilize the Flash-RAM storage exclusively for random lookups.

Thus, the processing time per index list now becomes 1 ms + 200/100 ms = 3 ms, and we can process 333 single-term queries/s on one computer. So theoretically, to meet the throughput requirements we would need only 3 computers for a normal load of 1,000 single-term queries/s or, actually, 6 computers for 1,000 queries/s with 2 keywords per query. However, we still need 157 computers for storing the complete index, with each computer costing \$1,000 + \$500 + \$800 = \$2,300. This builds up to a total cost of 157 * \$2300 = \$361,100. Once again, we configure the total system with a number of clusters of this kind. Note, though, that in this case we can do with a much smaller number of such clusters, since the replica clusters are needed solely for storage/availability/reliability and not for throughput as was the case for the previous architectures. Hence, assuming 5 clusters, leads to a total cost of \$361,100 * 5 = \$1.8 Mio. This is about 20 times cheaper than Alexander but still more expensive than Aeneas.

Furthermore, alternative Anakin configurations are possible, riding on the coexistence of disk, RAM, and flash-RAM memory technologies. Note that Anakin stores in the flash-RAM of each cluster the complete 10 TB index. Alternatively, we can opt for maintaining a single copy of the complete index in the Flash-RAM of the computers of all clusters. This still provides tolerance against failures since the disks in each computer can be used to store replicas of the complete index. Therefore, Anakin can now be configured with 10 clusters, each with 16 computers, yielding a total of 160 computers, each costing \$2,300. Thus, the total cost builds up to 160 * \$2,300 = \$368,000. This represents an improvement of a factor of 2 compared to Aeneas.

Of course, this is a very crude calculation, with many underlying assumptions that may not hold in practice. But the potential savings compared to the Alexander architecture are so dramatic that even if many estimates are off by an order of magnitude, we would still arrive at much lower cost than Alexander. The most critical assumption that might render our design calculations questionable if not invalid is the issue of load balancing. We discuss this in the following section.

6. LOAD BALANCING

Two key assumptions made so far is that the query-keywords distribution and the index-list size distribution are uniform. Obviously, this is not the case in the real world. We now study the three architectures under the prism of skewed keyword popularities and non-uniform index list sizes.

Alexander

Our first conclusion is that Alexander is immune to query-keyword skewness alone. This architecture delivers 1,000 queries/s per cluster, engaging all computers in a cluster and regardless of which keyword is involved, since each cluster stores the complete index and can thus serve any query. Things, however, change when index list size distributions are not uniform. Specifically, the query processing cost for queries involving terms with long index lists changes. Assuming that such long index lists are 10 times longer than the average size (which was set in Section 2 at 200 KB) the list-processing cost for such a long list in a single computer is now 2 s (compared to the previous 200 ms) and thus the list-processing cost in a cluster of 3,000 computers is now 2 s / 3,000 $\approx$ 0.67 ms. Therefore, the complete query processing cost now becomes 1.67 ms (1 ms startup cost plus 0.67 ms for list-processing at each computer). With this in mind, a cluster can now serve 598 queries/s. In order to match our peak load of 10,000 queries/s, Alexander now needs roughly 17 clusters, each with 3,000 computers, for a total of 51,000 computers and a total cost of $51 Mio. Of course, this is an upper bound, since not all queries involve the terms with the longest lists. Correlations between term popularity and index-list sizes, thus, play a key role here. If the most popular terms have small index lists then the cost calculated in Section 3 for Alexander (i.e., $30 Mio.) would be approximately correct here too. When the most popular terms have the longest index lists, the expected cost would be closer to the $51 Mio figure.

Aeneas

Aeneas, on the other hand, seems at first sight more vulnerable to skewed query-term distributions. This is due to term partitioning, which stores a term's index list to (the disk of) a single computer capable of serving a mere 5 queries/s. However, recall that Aeneas utilizes clusters of 200 such computers to reach 1,000 queries/s and then employs 10 such clusters to match peak load requirements of 10,000 queries/s. The pivotal observation is that each cluster (of 200 computers) has enough disk space (200 TB) to store 20 copies of the complete index (10 TB). And thus, Aeneas with its 10 clusters can store 200 complete copies of the total index. The key to scalability within Aeneas lies in the intelligent exploitation of this large storage space. Retaining the fundamental properties of Aeneas (disk exploitation, top-k threshold algorithms, and DHT-based term partitioning) there is a large number of intra-cluster and inter-cluster configurations that exploit the available disk space and can meet the challenges of non-uniform term popularities and index list sizes. A detailed discussion of these lies outside the scope of this work. Here we consider only an indicative, extremely challenging scenario for Aeneas.

Let us assume that 1% of the terms (i.e., 10^5 terms) are the most "popular" terms and that these are mentioned (uniformly) in 99% of the queries. Further, assume that these popular terms have the longest index lists (which are in turn 10 times greater than the average index list, i.e., each being 10 MB). To store the index lists for all popular terms requires 10 MB * 10^5 = 1 TB of extra disk space (i.e., one extra disk of 1 TB per computer). This bumps up the computer cost in Aeneas to $2,000.

Now, of the 10,000 queries/s (at peak load), 9,900 involve (uniformly) 9,900 terms (out of the 10^5 popular terms). The query processing cost is dominated by the list processing of the 10 MB index list, which includes 2 Mio. postings, the processing of which takes (1 ms per 1,000 postings and thus) a total of 2 s. Thus, at best each computer can yield 0.5 queries/s. Utilizing top-k threshold algorithms as before, we can cut down the list processing cost to 10%, yielding a query processing cost of about 200ms and improving the per-computer throughput to 5 queries/s. Thus, to reach our target of 10,000 queries/s we need to employ a total of 2,000 computers. To account for two-term queries we would thus need 4,000 computers at worse. These 4,000 computers already have great redundancy for the index lists of the popular and unpopular terms. The total cost for Aeneas then becomes $2,000 * 4,000 = $8 Mio, better by a factor of over 6, when compared against Alexander.

Anakin

Again, there are several alternative Anakin configurations that can deal effectively with nonuniform term-popularity and index list size distributions; for example in the spirit of [19]. Here we present a simple one that is indicative of Anakin's potential.

As before, in Anakin each computer has 64 GB of Flash-RAM, and 160 computers are needed to store one copy of the index lists of all terms in Flash-RAM. Note that the operating assumption is that the queries for the popular terms, involve terms uniformly from the set of popular terms. Thus, the 9,900 queries/s that arrive at peak load and involve popular terms, involve with high probability 9,900 distinct popular terms (from the 100,000 popular terms). Therefore, it is possible to forward each such query to a different computer, and in particular the one which happens to store the index list for the query's term in its Flash-RAM. Thus, each computer exploiting the cheap random accesses of Flash-RAM can cut down the disk access cost needing to scan only about 1% of the index list of a term (that is, only 20,000 of the 2,000,000 postings) requiring 1 ms for each 1,000 postings. Thus, the list processing cost is approximately 20 ms, for a total of 21 ms including startup costs. Hence, each computer can serve about 50 queries/s. To reach our peak load of 10,000 queries/s it is enough to employ a total of 200 computers, each costing $2,300, for a total cost of $460,000. To handle two-keyword queries, we employ 400 computers with the total cost becoming $920,000. This represents an improvement of a factor of about 9 compared to Aeneas.

7. INDEX CONSTRUCTION AND MAINTENANCE

For the discussion on index construction and maintenance, we need to distinguish two cases: the document-based partitioning of index lists by Alexander, and the term-based partitioning of index lists by Aenas and Anakin. The latter two architectures only differ with respect to their local data storage and degree of redundancy, which is a secondary issue as far as index building is concerned. Thus, we focus on the 10-cluster, 40-nodes-per-cluster Aeneas variant as a representative architecture for term-based partitioning.

Regarding the index construction we assume that all document are initially conceptually present at some "super-node" (e.g., a subset of the computers in an Aeneas DHT); i.e., we consider the crawling process that downloads Web pages as an orthogonal issue not considered here. We further assume that this super-node has only downloaded the indexable content, but not yet performed the actual extraction of index postings (i.e., *(term, page, score)*-triples). In fact, we want to leverage our computational powers to speed up the indexing process far beyond the rate that the centralized super-node would be able to sustain.

Regarding index maintenance we assume for our calculations that within one year an additional 20 Bio. Web pages need to be indexed; i.e., roughly 50 Mio. documents per day — this includes both previously unseen/new pages as well as updates to existing pages. We further claim that instantaneous index updates are unnecessary; instead we claim that one update per day, e.g., during off-peak hours, meets the requirements of keeping the index up-to-date. (Notwithstanding this claim, certain highly popular and frequently changing pages like news portals could be re-indexed every few hours.)

As parts of both processes will potentially be limited by the available network bandwidth, we assume an intra-datacenter connectivity with a point-to-point bandwidth of 10 Gbit/s, which – for the sake of simplicity – is assumed to be exclusively utilizable at the full raw bandwidth.

Document-based partitioning

As the hash-partitioning of pages guarantees that (within one cluster) all index postings for one page reside on exactly one peer, it is straightforward for the super-node to assign the duties of indexing accordingly. Naively, in the 10-cluster setting illustrated in Section 3, we need to distribute $5 * 10^7$ pages to 10 nodes each (one target node for each page within each cluster), resulting in a network traffic volume of $10 * 5 * 10^7 * 500$ bytes $= 250 * 10^9$ bytes = 250 GB. (recall that we assume an average of 500 bytes of indexable content per document). If the network can be fully utilized at its nominal raw bandwidth, this would keep the network busy for 250 GB * 1 GB/s = 250 seconds. Even if - more realistically - we can effectively utilize the network only at 10 percent of its raw bandwidth, this would take less than an hour. Each node in a 3,000-node cluster will approximately receive 17,000 pages per day, which is small enough so that local indexing time can be ignored. So daily batches of index maintenance incur only a moderate network and processing load on the Alexander data center.

Term-based partitioning

For our architectures based on term-based partitioning, the situation is more complicated, because conceptually each new page has an impact on those (up to) 40 nodes that maintain the index lists for its 100 indexable terms in each cluster. So the whole page would have to be shipped to up to 40 nodes. This would increase the network and processing load for index maintenance by a factor of 40, rendering the Aeneas design critical if not practically infeasible.

Here the key is a smart choice of where the indexing of a page is performed. Our proposed solution is a two-phase strategy that uses the existing DHT infrastructure with its hashing capabilities to partition the pages across nodes. In the first phase we hash-partition new pages by page IDs onto a sufficiently large number of nodes, and it is these nodes that parse the pages and extract the index postings. The nodes can be chosen from the same network that handles the query workload; they merely need to provide a certain fraction of their memory and CPU capacities for low-priority background work. In the second phase, the nodes that have prepared the postings send them to the nodes that are responsible for the corresponding terms in the term-partitioned design, now using a hash function on term IDs. At the end of the second phase, all new index postings reside at the right target nodes, and need to be merged into the existing lists incurring the same local processing load that the Alexander design needs to sustain.

For an analysis, consider again that the super-node spreads the $5 * 10^7$ documents to 40 nodes of one Aeneas cluster, creating $5 * 10^7 * 500$ bytes = 25 GB of network traffic, which consumes 25 seconds if the network could be fully utilized for these transfers in a single burst or 250 s at 10% network utilization or still less than an hour with load throttling that uses only 1% network utilization in the background. Each of these 40 nodes now performs the indexing of 500,000 pages, which is not totally negligible, but feasible. Subsequently, each of the 40 nodes can disseminate its 500,000 docs * 100 terms * 5 bytes/entry = 250 MB of new index content to the final targets in all 10 clusters. This second phase creates a total network traffic volume of 25 GB, and just like in the Alexander design, the entire dissemination can be done in less than an hour.

Thus, even if we have ignored certain additional aspects (like failures among the 400 nodes that compute the index postings), index maintenance seems practically feasible under realistic constraints in term-partitioned systems like Aeneas and Anakin.

8. CONCLUSION

This paper has aimed to shed new light into the ongoing debate about whether distributed and peer-to-peer indexing are practically viable for ultra-scalable Web search. It has identified various interesting design points, in terms of partitioning strategies and storage technologies, and discussed their strengths and weaknesses. In particular, we have offered simplified but hopefully insightful analyses of the cost/performance ratios for various designs. The Anakin design, based on term partitioning and a combination of disk and Flash-RAM storage, seems to be a particularly intriguing option.

Obviously, the paper's analyses are not yet detailed enough for final conclusions. But we believe that they are indicative and encourage further research along these lines, in pursuing the quest for fully decentralized, scalable, and highly efficient Web indexing. The anticipated further growth of the Web and the load on search engines suggest that this is not only an academic exercise but will become a pressing issue in coming years.

9. REFERENCES

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, pages 179–194, 2001.

[2] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *SPIRE*, pages 304–315, 2006.

[3] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges in

distributed information retrieval (invited paper). In *ICDE*, 2007.

[4] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k at TREC 2006: Terabyte Track. In E. M. Voorhees and L. P. Buckland, editors, *Proceedings of the 15th Text REtrieval Conference (TREC 2006)*, pages 551–555, Gaithersburg, Maryland, 2006. NIST.

[6] M. Bender, S. Michel, J. X. Parreira, and T. Crecelius. P2p web search: Make it light, make it fly (demo). In *CIDR*, pages 164–168, 2007.

[7] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. P2P content search: Give the web back to the people. In *IPTPS*, Santa Barbara, US, 2006.

[8] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, 2007.

[9] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[10] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[13] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. In *CIDR*, 2007.

[14] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS*, pages 207–215, 2003.

[15] J. C. S. Lui, R. R. Muntz, and D. F. Towsley. Computing performance bounds of fork-join parallel programs under a multiprocessing environment. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):295–311, 1998.

[16] T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. Alvis peers: a scalable full-text peer-to-peer retrieval engine. In *P2PIR*, pages 41–48, New York, NY, USA, 2006. ACM Press.

[17] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72, 1997.

[18] S. Michel, M. Bender, N. Ntarmos, P. Triantafillou, G. Weikum, and C. Zimmer. Discovering and exploiting keyword and attribute-value co-occurrences to improve p2p routing indices. In *CIKM*, pages 172–181, 2006.

[19] S. Michel, P. Triantafillou, and G. Weikum. Minerva$_{\text{infinity}}$: A scalable efficient peer-to-peer search engine. In *Middleware*, pages 60–81, 2005.

[20] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, pages 348–355, 2006.

[21] S. Nath and A. Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN*, pages 410–419, 2007.

[22] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable Peer-to-Peer Web Retrieval with Highly Discriminative Keys. In *ICDE*, 2007.

[23] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[24] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB J.*, 7(1):48–66, 1998.

[25] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[26] T. Suel, C. Mathur, J. wen Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB*, pages 67–72, 2003.