I/O-Conscious Data Preparation for Large-Scale Web Search Engines

Maxim Lifantsev

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
USA
maxim@cs.sunysb.edu

Tzi-cker Chiueh

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
USA
chiueh@cs.sunysb.edu

Abstract

Given that commercial search engines cover billions of web pages, efficiently managing the corresponding volumes of disk-resident data needed to answer user queries quickly is a formidable data manipulation challenge. We present a general technique for efficiently carrying out large sets of simple transformation or querying operations over external-memory data tables. It greatly reduces the number of performed disk accesses and seeks by maximizing the temporal locality of data access and organizing most of the necessary disk accesses into long sequential reads or writes of data that is reused many times while in memory. This technique is based on our experience from building a functionally complete and fully operational web search engine called *Yuntis*. As such, it is in particular well suited for most data manipulation tasks in a modern web search engine and is employed throughout *Yuntis*. The key idea of this technique is coordinated partitioning of related data tables and corresponding partitioning and delayed batched execution of the transformation and querying operations that work with the data. This data and processing partitioning is naturally compatible with distributed data storage and parallel execution on a cluster of workstations. Empirical measurements on the *Yuntis* prototype demonstrate that our technique can improve the performance of external-memory data preparation runs by a factor of 100 versus a straightforward implementation.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

Web search engines such as AltaVista [3], Fast Search [2] and Google [11] are an indispensable tool for web surfers to access the information on the global Web. In the past two years, we have been working on the implementation of a full-scale web search engine that is based on a more general and powerful resource ranking model [16, 17] than Google's PageRank [19]. During this effort, we found that the data preparation process in a search engine poses a set of different requirements than traditional database applications have, and thus deserves development of new techniques to optimize its performance. This paper reports the results of this investigation.

To answer user queries efficiently, search engines in particular need to prepare an index, which is usually in the form of an inverted index file. That is, to a first approximation it associates each keyword with the list of pages in which the keyword appears. To construct an inverted index, a search engine needs to collect pages from the Web, parse them, assign an identifier to each page, and put the identifier of each page into the hit lists for all keywords that it contains. In this process, various types of data structures other than the inverted index itself need to be created. In addition more recently developed search engines such as Google [11] compute various scores for all web pages by performing a non-trivial iterative computation over the whole web linkage graph. Because of the sheer data volume involved, most of the data tables, including the inverted index and the data for page score computation, can not fit into the main memory of the processing servers: Modern search engines presently index over two billion web pages [11] and manipulate terabytes of data. As a result, use of efficient external-memory

algorithms that minimize the disk access overhead and use of parallel processing on clusters of workstations is crucial for achieving acceptable performance for search engine data preparation.

From a database system design standpoint, the key difference between web search engines and standard database systems is that the consistency requirement in response to updates is much weaker for a search engine. The general assumption of a search engine is that keeping a larger and more recent snapshot of the Web is more important than maintaining an absolute up-to-date view of a (inevitably) smaller portion of the Web. As a result, a search engine does not need to react to page updates immediately, and is free to incorporate them in any order as long as they are completed within a period of time, such as few days.

From the implementation experience of the 124,000-line *Yuntis* web search engine prototype, we have derived a general implementation approach for external data manipulation that we believe is equally applicable to other data-intensive applications with similar data volume, processing, and performance requirements and properties. To reduce the disk access overhead, we apply a data-driven processing principle to minimize the I/O costs for external-memory data to all major data preparation tasks in the search engine prototype. That is, once we paid for I/O cost for a group of external data items, we try to use them as much as possible before they have to be vacated from RAM. In addition we try to organize most of the necessary disk I/O into a set of long sequential buffered reads and writes instead of a larger set of smaller random accesses, that would incur a much greater number of disk seeks. We achieve this by physically splitting in a coordinated fashion closely related external data structures into disjoint partition groups that are small enough to fit into RAM. We also logically arrange all processing performed on external data into a set of sequences of simple procedures that only need data from one such partition group. Then we organize the processing in a batched way: The input data for procedures over a data partition group is queued on disk. Then long enough (if possible) queues are executed by sequentially bringing the partition group's data into and out of RAM for the time the queued procedures are being run. In addition we perform this data and processing partitioning over a locally-connected cluster of shared-nothing workstations to exploit inter-operation parallelism. Use of an efficient non-preemptive uni-threaded data-driven process model allows us to exploit application concurrency inexpensively both at run and coding time.

The rest of the paper is organized as follows. We review related work in Section 2. Section 3 presents the main distinguishing features of web search engines and their common data manipulation pattern. In Section 4 we show how the data-driven processing technique exploits the data manipulation pattern for high performance search engine data preparation. We illustrate this technique with an example, discuss its various design issues, and describe how cluster of workstations and an event-driven processing model can be used to further improve the performance. We then report the results of a performance evaluation study based on the fully operational *Yuntis* search engine prototype in Sections 5. Finally we conclude this paper with a summary of major research contributions and an outline of future work.

2 Related Work

There has been a substantial amount of recent research on the design and efficient implementation of various features of web search engines. In particular, Ribeiro-Neto, et al [20] described an inverted index construction scheme carefully optimized for clustered execution. Melink, et al [18] proposed a distributed index construction method, which applied explicit disk, CPU, and network communication pipeline. Hirai, et al [14] analyzed different design aspects of a repository system for generic web page storage, update, and retrieval. Haveliwala [12] described a disk-efficient non-distributed method to compute PageRank [19] for large web linkage graphs. Stata, et al [22] showed a design of how to construct a filtered database of "interesting" words for all web pages. Bharat, et al [5] described their experience building a server that constructs and queries a compact representation of forward and backward linkage among a set of web pages.

The common property of these efforts is that each concentrates on one or few search engine data preparation tasks and develops an efficient algorithm tailored for them. On the other hand, not much open information is available about detailed integrated design of all processing tasks in a modern large-scale (commercial) web search engine. Brin and Page [6] briefly describe several design decisions made when building the early prototype of the Google search engine [11]. The approach taken is loosely putting together algorithms each explicitly optimized for its specific data manipulation task.

In the area of disk access optimization there has been work on disk data placement [24], smart disk head scheduling [21], data replication on one or many disks [1, 26] to reduce (read) seek times. These techniques usually work at the disk driver or OS level and can not automatically perform well in all cases. Disk data prefetching schemes can only mask disk access when the data processing time is larger than the I/O time. Similarly, data striping or use of multiple independent disks simply produces a virtual disk with better seek and/or throughput parameters than the real disks it is made of. Our technique is orthogonal, thus complementary, as it works at the application level to increase temporal access locality and proportion of se-

quential disk accesses.

The closest analogs of our approach are the batch filtering technique [10] for bulk simultaneous searching in an out-of-core DAG data structure from the area of external-memory algorithms [23] and the data-driven processing ideas for volume rendering of Yang and Chiueh [25]. Both methods rely on reuse of data items (pre)fetched from disk as much as possible by delaying and performing all necessary processing on them in one shot.

Our work focuses on the design of a general method for efficient I/O-conscious bulk manipulation of external-memory data structures. The method is mechanically applied to improve performance of the natural formulation of data manipulation algorithms, eliminating the need to optimize each task individually. Thus, it can be easily used with all data preparation tasks within a modern web search engine including text indexing, linkage construction, incremental graph score computation, and extraction of corpus phrases and document key terms. We also believe that the proposed approach similarly applies to other tasks where large sets of updating and/or querying operations have to be executed efficiently over partitionable external-memory data structures.

3 Data Preparation in Web Search Engines

To provide fast and accurate response to user queries, web search engines [11, 3, 2] pre-compute various types of data structures from the pages collected on the Web [6, 4]. Examples include the inverted index, the web page linkage graph, the list of URLs previously visited, results of text classification, etc. Sometimes one data structure is used together with another data structure to improve the querying service efficiency. For instance, page "quality" scores can be embedded into the inverted indexes to filter out undesirable pages early on [6].

3.1 Data Preparation Tasks

Data Acquisition When a web crawler gets a URL, it needs to check if the URL has been already visited. If not, the crawler marks it as "already visited" and schedules it to be crawled in the future. To mark a URL, the crawler typically would store the URL name string, allocate an internal identifier for it, establish the mappings between the two, and add an initial information record about the URL.

Standard Data Preprocessing To update the inverted index of web pages, when the crawler knows that a given word occurs in certain places of a crawled web page after parsing it, it must determine the internal identifier of the word —possibly adding the necessary records if the word appears for the first time— and appropriately update the hit list for the word with these word occurrences.

To build internal data structure about web page linkage, when the crawler encounters a hyper-link from one page to another, it must first locate the internal identifier for each page's URL, adding the new URLs to the databases, and update the lists of forward and backward links associated with the two URLs.

Score Computation Web page score (rank) computation according to the PageRank method [19, 6], Hubs and Authorities approach [15], or the voting model [16, 17] is performed in multiple iterations. At each iteration the algorithm usually goes through all the pages. For a given page, the algorithm first retrieves the current score values of the page and the list of pages that it points to and/or that point to it, and then adjusts the score values of these pointed and/or pointing pages, according to the page's current score values. Some statistical data such as the amount of overall score changes between two consecutive iterations are also usually computed.

These data preparation tasks constitute the bulk of pre-query work in a typical web search engine. They usually can be thought of as composed of execution of simple procedures (of a relatively small number of types) that each given some input data items read and/or update some (parts of) certain records in few data tables and possibly initiate execution of other such procedures. The instances of each kind of these procedures that have to be executed to make all search engine data structures consistent as a whole after a large batch of updates due to crawling usually touch multiple times a substantial portion of the external data tables they are working with.

3.2 Data Manipulation Approaches

Given that modern search engines cover over one billion of web pages [11], most types of data structures web search engines are working with are in external memory even for a relatively large cluster of modern workstations.[1] In some cases, even individual data items such as lists of occurrences of frequent words or lists of back-links to popular web pages do not fit into RAM of affordable workstations. As a result only linear-time and disk-aware data manipulation algorithms can be used. The actual implementation of these algorithms must have small associated constant factors because, for instance, a performance difference of only a factor of 100 will turn a reasonable task that runs for a few days into an impossible job that would have taken over a year to complete. In particular, the implementation must avoid performing unnecessary disk accesses, especially seeks.

[1] For example, basic information about each encountered URL including its name can occupy 128 bytes on average in our prototype. Thus, just this basic information about one billion URLs will require around 128GB of storage.

A naive way to manipulate the data is to follow a control-driven approach. That is, to perform most of data updates as soon as it can be done. For example, whenever a web page is collected, it is first parsed, then the associated inverted index entries are modified, and other derived data structures such as web page linkage graph are changed accordingly. In this approach, each web page update may trigger a large set of disk I/Os. In general, this control-driven approach, although conceptually simple and straightforward to implement, leads to very inefficient use of the disk resource, because the data brought in by the disk I/Os associated with the processing of one data modification cannot necessarily be reused by the processing of subsequent updates.

One can somewhat improve the performance of the control-driven approach by using (virtual striped) disks with better characteristics [21], using more RAM or fitting more data into RAM with smart encodings [5, 22], or start moving away from the purely control-driven computation by explicitly exploiting data patterns and access locality that exists in a particular workload [13, 8], or abandon the control-driven approach in favor of developing very task-specific algorithms and data structures [12, 20, 6]. But these methods either do not provide a radical performance improvement or are hard to apply mechanically to a wide set of data preparation tasks.

In light of the failing of the control-driven approach, a data-driven approach is considered more appropriate. In this case, we perform elementary manipulation procedures over external-memory data only when the data they need is (very likely to be) in main memory. This is achieved by delaying execution of procedures and batching their input into queues associated with external-memory regions the procedures are going to work with. In addition, at some points in time the system will bring into RAM these external-memory data regions one by one and activate all the procedures that have been queued on these data regions. Thus, these data regions are reused multiple times and the associated disk I/O overhead is amortized over multiple operations — we strive to trigger the computation on an external region only after a lot of procedures have been queued to it. Essentially, a data manipulation procedure is triggered by the availability of the data it needs in memory. Note that, in addition to pure data-flow requirements there is usually still some higher-level control-flow scheduling to be obeyed, for example, breadth-first crawling might be required to be done not arbitrarily, but by frontiers of new pages which are one link away from the set of currently known pages.

This data-driven approach relies on batching the procedures that require the same data and performing them in one shot when the data is in memory. A well known problem with operation batching is that it increases operation latency. Fortunately, the latency of individual procedures is not a concern in the case of a web search engine. The performance goal of a search engine's data preparation algorithm is not to complete the data update with respect to each web page update as quickly as possible, but to finish all the data updates for large sets of crawled web pages as soon as possible.

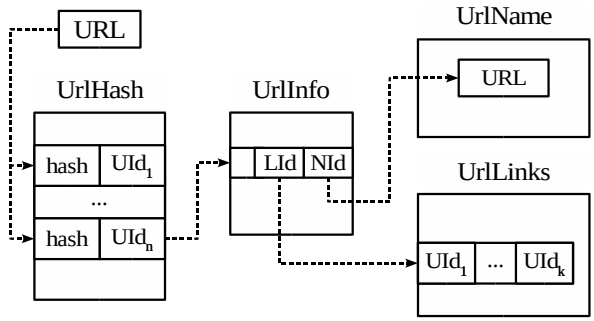

Figure 1: Data table structures used during web page back-linkage construction process.

4 Data-Driven Data Manipulation

Let us start with a concrete example to illustrate the typical processing flow for data preparation in large-scale search engines, and then present the specific implementation technique we used to optimize the external-memory data manipulation process.

4.1 An Example Scenario: Back-linkage Construction

The goal of this task is to construct the backward linkage information for a set of web pages. Assume that we are working with the following data tables (see Figure 1):

`UrlHash` is a set of fixed-width URL name hash values and internal URL identifiers (`UId`s) and is used for locating data by a URL name. It is organized in such a way that its records can be searched by a hash value or added in $O(n \log n)$ (amortized) time or less.

`UrlName` is a memory heap-like data table that holds variable-width strings of URL names that can be accessed by fixed-width pointers and is used to store the URL names.

`UrlLinks` is a data table similar to `UrlName` and contains lists of `UId`s for the pages that point to a particular URL. This table represents the URL linkage graph.

`UrlInfo` is an array of fixed-width URL information records indexed by `UId`s, and in particular contains pointers into `UrlName` and `UrlLinks` structures. This structure ties the others together and maps a `UId` to all data stored about that URL.

Given the URL identifier of the linking page and the URL of the linked page, the back-linkage construction procedure

1. Searches `UrlHash` for a set of `UId`s with the same hash value as that of the URL of the pointed page,
2. Verifies if any of these `UId`s really correspond to the pointed page's URL by reading and comparing the URL names associated with these `UId`s by following the pointers from `UrlInfo` to `UrlName`,
3. Creates a new `UId` and inserts appropriate new elements into the tables if it could not find a match, and
4. Adds the `UId` of the linking page to the `UrlLinks` list of the linked page by following (and sometimes updating) the `UrlLinks` pointer in the `UrlInfo` entry.

The above procedure corresponds to the work for one pair of linking and linked pages, and touches only on the portions of these tables that are related to the hash value of the linked page's URL. Instead of executing the above procedure as soon as it is possible for each hyper-linked page pair, we choose to work on many such page pairs in batches, trying to do it so that once a piece of data table is brought into memory from disk, it is reused as many times as possible. More specifically, we partition the four data tables in a coordinated fashion, so that each step in the above procedure can proceed using the data from only one group of data table partitions. In addition, when the data table partition group that a procedure requires is not memory-resident, the procedure will be attached to the data table partition group and will get executed once the data table partition group is available in memory. Because we usually wait for multiple procedures to get attached to a data table partition group before executing them, this data-driven approach enables batching of procedures that require the same data table partition group, thus greatly decreasing the disk I/O cost. Moreover, the disk access pattern is no longer dictated by the control flow of the data manipulation algorithm. As a result, most disk I/Os can be sequential accesses, which further improves the disk bandwidth utilization.

Our data-driven approach is mainly oriented towards optimizing disk reads and is not concerned with disk writes because no synchronous disk writing of small data items is required in search engine data preparation. Hence, disk writing is left to standard file cache management in the underlying operating system as we use regular OS files for data storage. However, explicit long sequential file writes occur frequently as we will see shortly.

In the case of back-linkage construction, to apply the data-driven processing approach we first split the four data tables according to ranges of the hash values of the URLs whose information is placed into the

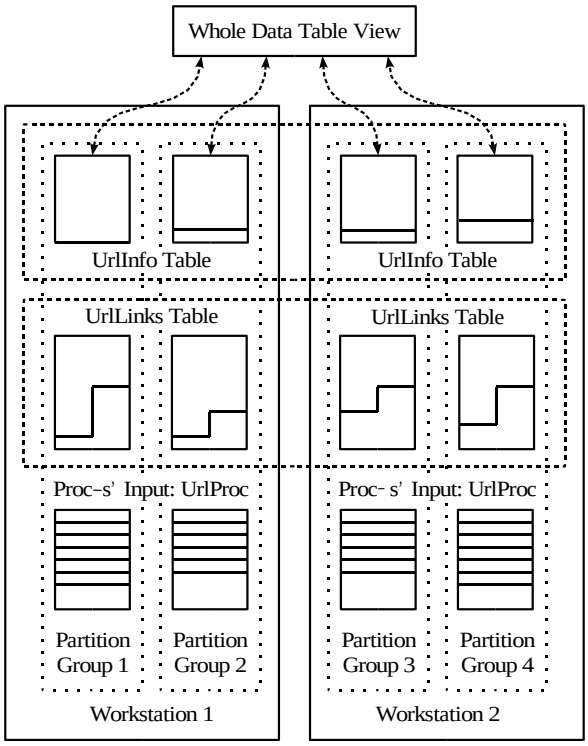

Figure 2: Coordinated partitioning of related data tables and the input data for transformation procedure batches over these data tables. Procedures in each partition group need the data only from the table partitions of that group.

records of table partition groups — Figure 2 provides a simplified illustration with `UrlHash` and `UrlName` tables omitted. That is, `UId`s stored in partition i of `UrlHash` refer to the records only in partition i of `UrlInfo`, which in turn contains pointers only into ith partitions of `UrlName` and `UrlLinks`. New data table records are always created within a certain partition group determined by the hash value of the URL name, and existing records cannot move across partitions once created. A portion of `UId` bits is used as the partition number, while the remaining bits are used for record identifiers within each partition. If pointers into `UrlName` and `UrlLinks` are not stored anywhere else except `UrlInfo`, then they can be partition-local, otherwise their portion too has to be used for the partition number. The number of partitions is such that at the targeted data set size one partition of `UrlHash`, `UrlName`, and `UrlInfo` can still fit into main memory simultaneously, with enough space left for caching a portion of a `UrlLinks` partition.[2]

Secondly, we introduce a new table `UrlProc`, which is partitioned the same way as other tables but is organized as a queue of records containing URL identifiers

[2]We do not expect to hold a complete `UrlLinks` partition in memory because individual link lists for few popular URLs could be very large since the number of links follows the power-law distribution [7].

and URL names of a hyper-linked page pair. When a back-linkage procedure for a hyper-linked page pair is ready to run, we do not start it immediately. Instead, we queue the input data into a `UrlProc` partition based on the hash value of the linked URL in the hope to batch the execution of multiple such procedures later on. When enough procedure input pairs have been accumulated or when there is no other task that can proceed, we execute all the procedures partition by partition: for a batch stored in the ith `UrlProc` partition each procedure within is executed as follows. We sequentially read into memory ith partitions of `UrlHash`, `UrlName`, and `UrlInfo`. We then execute the procedures whose input tuples are in the ith `UrlProc` partition by sequentially reading in its contents. These procedures operate only on the data in ith partitions of `UrlHash`, `UrlName`, and `UrlInfo`, which are in memory, and on the data in the ith partition of `UrlLinks`. With an appropriate organization of `UrlLinks` table we will be mostly touching the cached heads and tails of few link lists in `UrlLinks` during this execution of procedures. After executing all the procedures associated with ith `UrlProc` partition, we unload the resulting modified partitions of `UrlHash`, `UrlName`, and `UrlInfo` to disk by sequential writing, possibly after performing some compression or clean-up operations on them. In the case when there are many tables like `UrlProc` that have enough buffered data, we can execute the procedures for example table by table and partition by partition within each table. The main requirement is to execute the procedures one partition at a time to take full advantage of the memory, preferably doing it only when the input for enough procedures has been accumulated, and keeping the size of all buffered input data not too large.

4.2 Analysis of the Data-Driven Approach

4.2.1 Analytical Performance Analysis

Overhead The performance overhead of the data-driven data manipulation approach lies in additional packing, unpacking, reading, and writing of the input data of the procedures whose execution is delayed and batched. The CPU and disk I/O cost of this is linear with respect to the number of procedures executed and only sequential disk I/O is required for this. The table partitioning adds only a small constant overhead to direct each table's access to the appropriate partition.

If we are forced to do very short batches of data manipulation procedures to meet some overall computation dependencies, then we revert to the performance of the control-driven strategy.

Benefits The main benefit of the proposed data-driven strategy is that it guarantees in-memory access for all data in one class of data tables and much improved data access locality for the other tables during the data preparation process. In addition, accesses to the first class of data tables are serviced by sequential disk reads and writes, hence utilizing the disk bandwidth much more efficiently.

Assume that we have $n = 3$ data tables that hold information about 500 million objects with $d = 10$GB of data totally. There are $m = 100$MB of RAM available for data and file cache, and one disk with average seek time of $s = 9$msec and a sustainable data transfer rate of $r = 30$MB/sec. Assume we need to perform $o = 50$ million data manipulation procedures each of which touches all three tables exactly once and requires input data of size $i = 100$B per procedure on the average.

Under a random access pattern, the control-driven approach will lead to $m/d = 1\%$ file cache hit ratio. Therefore each procedure will require $(1-m/d)*n*s = 26.7$msec time for disk seeks alone. If the data tables are decomposed into $d/m = 100$ partitions, each of which fits into RAM, then the total disk data transfer requirement will be $2*i*o+2*d$ for writing and reading of the procedures' input data and the data tables. Thus, the disk I/O time per procedure in the data-driven approach will be only $(2*i*o+2*d)/o/r = 0.02$msec, which is more than 1300 times better. Here we assume that each disk I/O in the data-driven approach is a sequential one and thus the disk seek delay is negligible.

4.2.2 Optimization

Task-specific Optimizations Some task-specific optimizations can be embedded into this general data-driven strategy. For example, a locality-preserving hash function that exploits existing data patterns, such as high host locality of URLs that are close in the linkage graph, can be used to achieve better CPU and/or file cache utilization or to reduce intra-cluster communication [8].[3] Compact data encoding can be applied to individual data table partitions because its construction and updating can be naturally included during the loading and unloading of a data table partition. We can also periodically perform compaction of external-memory partitions of data tables like `UrlLinks`. When this compaction changes the pointers needed to access the records in the compacted partition, we can update these pointers without much additional cost because they are typically stored only in the respective partition of a data table like `UrlInfo` and all the data needed for performing such updates fits into main memory.

Exploiting Parallelism with a PC Cluster Because each batch (partition) of data manipulation procedures is independent of others, multiple batches

[3]The downside of this technique as our experience indicates is that it might increase unevenness of processing load on different cluster nodes, thus reducing the effectiveness of using a cluster. Also the benefits of this locality during crawling are not significant because URLs from the same host are not necessary crawled close in time when crawling is guided by estimated importance of the pages [9].

can be executed in parallel on a shared-nothing cluster of PCs connected with a high-speed local area network. If the data table partitions associated with a batch are placed on the same cluster node —see Figure 2, then the only additional cost of parallelism is the intra-cluster communication overhead due to the messages transferring the input data for procedures that are to be executed on a node different from the one where they were initiated. Since no result value is needed to be returned for these messages, they do not incur any significant synchronization delays among the nodes, because synchronization by their completion might be needed only when we finish executing all batches of the same type on a node. Other than that, execution of the procedure batches can proceed completely locally and independently on each node.

As the experimental data in Section 5.2 will show, data partitioning based of object name hashing does very well at dividing the work among cluster nodes evenly (or proportionally to each node's performance). Thus, no dynamic re-balancing transfer of work and data is usually required among cluster nodes to achieve high utilization of the cluster as a whole.

Efficient Data-Driven Process Model To maximize the CPU utilization efficiency, the data manipulation is structured as non-blocking with respect to the following I/O operations: transfer of input data for procedures among cluster nodes, other request/reply communication between search engine components on (different) cluster nodes, HTTP communication with web servers, and local disk reads and writes. In addition, to exploit spare CPU cycles we support execution of a large number of potentially concurrent activities. This is achieved by using one main control thread that performs descriptor polling and user-level work scheduling, non-preemptable small work units without I/O blocking, non-blocking network and disk I/O, and call/callback interfaces for potentially blocking requests instead of traditional call/return interfaces. This approach avoids the performance overheads associated with threads: kernel scheduling, context switching, stack and task data structures allocation, synchronization, inter-thread communication, and thread safety issues.

4.2.3 Discussion

Conditions for Application The three prerequisite conditions that allow the successful application of the data-driven approach to data preparation in search engines are that

- The scale of data table sizes needs to be known in advance to determine the number of partitions.
- A good hash function that can decompose data tables into approximately equally-sized partitions must be used.
- Each type of data manipulation procedure must use only the data from one partition group of data tables that are all partitioned in the same way.

Our experiences show that the first two conditions that guarantee that each data partition group will actually fit into the available memory are relatively easy to meet. However, the third condition requires additional attention. If a data manipulation procedure does not satisfy this requirement, it needs to be "split" into simpler ones that do satisfy this requirement. For example, assume we also wish to construct forward linkage information in `UrlFLinks` table in the same way as backward linkage is filled into `UrlLinks` table in the earlier example. Instead of augmenting the previously used data manipulation procedure, we have to introduce a second data manipulation procedure and a separate queue `UrlFProc` for its input tuples. Given two `UId`s the new procedure adds the second one to the list of forward links in the `UrlFLinks` record corresponding to the first `UId`. This new data manipulation procedure can be initiated as soon as the `UId` for the linked document is known. The splitting of the augmented version of the procedure into the new one and the old one with initiation of the new one is needed because updating forward linkage information touches data for the linking URL only and can be done only after the `UId` of the linked URL is known, whereas construction of the backward linkage information touches data for the linked URL only and determines its `UId`. This splitting of computation procedures adds a small level of additional overhead for enqueuing and dequeuing of the input data for the additional types of procedures.

The extra condition for general applicability of the data-driven approach to cases other than web search engines is that the throughput performance of a large volume of simple data transformation and/or querying operations is to be optimized when these operations work with external memory data sets without much automatic access locality.

Universality An important property of the proposed data-driven approach is that it does not place any substantial restrictions on the structure of data tables and how they are stored on disk. The only requirement is that the partition number must be embedded into the identifiers in the used object identifier address space. Actually, the property of loading some table partitions into main memory for the time they are actively accessed gives one an additional freedom of how the data in these table partitions can be organized. A demonstration of the universality of the data-driven approach is that in certain cases it actually leads to improved performance for a simple increment propagation algorithm that operates on a memory-resident sparse graph. The reason for this improvement is that the approach enhances the access locality and thus takes better advantage of CPU cache, in the same way

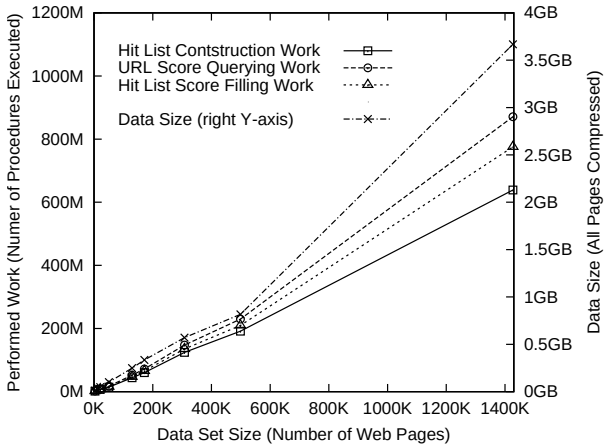

Figure 3: Linear growth of handled data and work performed in different workloads when data set size grows.

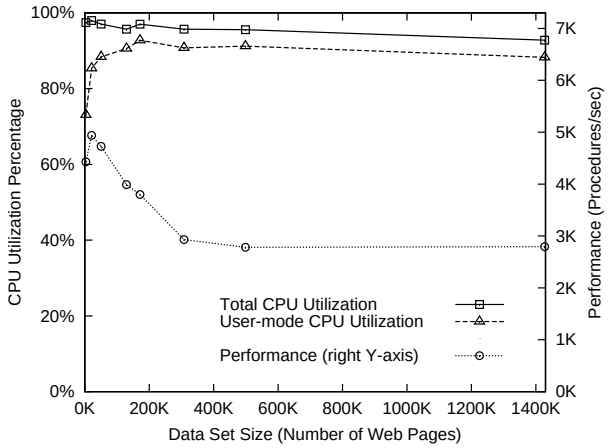

Figure 4: CPU utilization and Performance when constructing hit lists for increasing data set sizes.

as it can make more effective use of memory and OS file cache to access disk-resident data.

5 Performance Evaluation

5.1 Implementation

We have embodied the techniques described in the previous section in a fully operational search engine prototype, *Yuntis*, which is implemented in C++ under the Linux OS. The implementation consists of 871 files of 3.9MB total size (758KB when compressed with `gzip -9`) that contain 124,000 lines of code in 163 logical modules.

The latest version of the prototype is accessible at `http://yuntis.ecsl.cs.sunysb.edu/`. The largest current data set is based on 9 millions of crawled web pages. *Yuntis* utilizes a new model for assessing web pages' relevance and quality [16, 17], which subsumes and improves on the Google's model [6, 19]. The prototype is running on a cluster of four Linux PCs that are connected via a 100Mb/sec full-duplex switch and each have one Pentium III 650MHz CPU with 100MHz bus, 256MB of RAM, and use one EIDE Maxtor 98196H8 disk for data (9msec average seek time, 81GB capacity, and 46.7MB/s maximal media transfer rate).[4]

The flexibility of the proposed implementation techniques and the object orientation of our C++ implementation of communication and database primitives allow us to experiment with adding or modifying different architectural features, data tables, and data processing methods in the prototype quickly, while providing efficient network and I/O performance.

[4]Modern SCSI disks such as Maxtor Atlas 10K III only offer reduced 4.5msec seek time and slightly higher media transfer rate of 55MB/s for more than threefold price increase if we compare one disk installations.

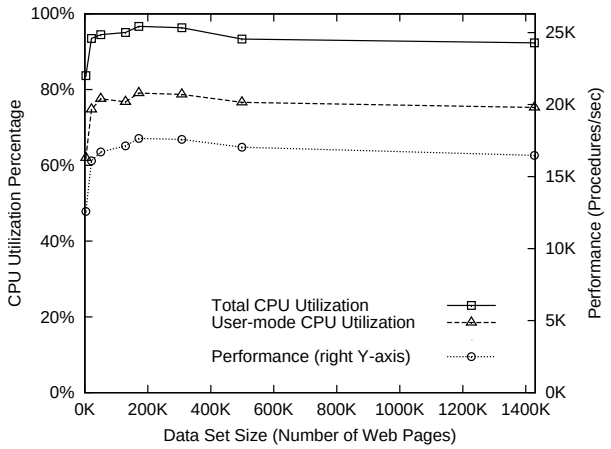

Figure 5: CPU utilization and Performance when querying URL scores for all hits for increasing data set sizes.

5.2 Results and Analysis

We have performed a set of experiments demonstrating effectiveness of the proposed data-driven technique of coordinated partitioning and batching for different search engine data manipulation workloads. The experiments show high CPU utilization and same high performance level when handling data sets primarily located either in RAM, or on disk. Direct comparison with straightforward control-driven implementation shows significant performance improvements for both RAM and external-memory data sets. Low-level measurements of disk read requests indicate much higher proportion of fast reads without seeks when our data-driven technique is compared with the control-driven one. Since our technique primarily works by increasing access locality and reducing the number of misses in RAM OS disk (and CPU) caches, the absolute sizes of the data sets used in the experiments are not as important for proving effectiveness of the approach, as their sizes compared with the amount of

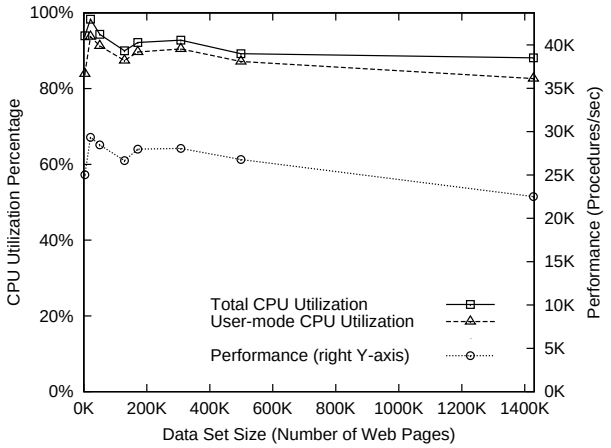

Figure 6: CPU utilization and Performance when filling URL scores for all hits for increasing data set sizes.

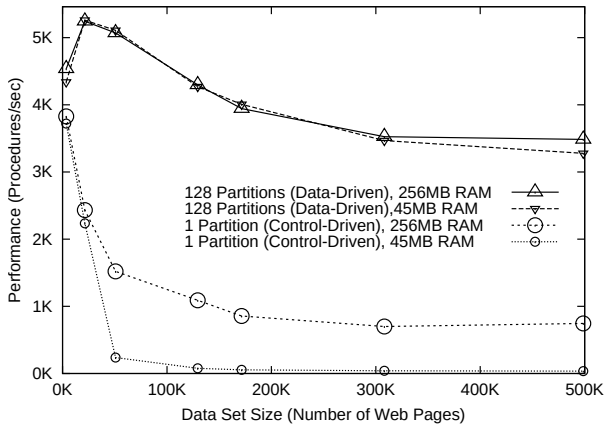

Figure 7: Performance comparison between data- and control-driven approaches when constructing hit lists for different data set and RAM sizes.

RAM available for the OS disk cache.

5.2.1 Effectiveness of the Data-Driven Approach on Various Workloads

Figure 3 demonstrates essentially linear growth with respect to the number of crawled web pages of the total size of compressed web pages and the required number of data transformation procedures in the three different data manipulation workloads we have chosen. Figures 4–6 provide the graphs for the total CPU utilization percentage for the application and the OS, the CPU utilization percentage for the application alone, and the performance in terms of the number of procedures executed per second when running the three workloads over increasingly larger sets of web pages.[5] In all workloads the data tables are partitioned based on hashing of the URL and word name strings into 64 and 128 partitions per each cluster node for URL- and word-related data respectively.

The procedures of hit list construction workload (see Figure 4) resolve a word name into its identifier and add the information about its occurrences in one document into its hit list, while adding the needed records for newly encountered words. In this workload the performance somewhat reduces while CPU utilization remains high due to the need for sequential disk I/O when the workload's data set becomes mainly disk-resident. In the URL score querying workload on Figure 5, each procedure gets a URL score value for a particular URL identifier via an array lookup and initiates execution of a hit list score filling procedure. The latter step consists of batching the input data for the procedures into an appropriate partition on disk that might be on another cluster node. A procedure of the hit list score filling workload from Figure 6 consists simply of changing the score fields in an existing hit list data structure that is accessed by addresses. The performance decline for the last data point is due to a peculiarity of the implementation used for the measurement: At this data size the processes on some cluster nodes sometimes had to receive and batch procedure input data from the other nodes, which took a portion of CPU resources.

Each of the three workloads require an essentially constant amount of CPU resources for OS tasks, but the amount is different for different workloads. Results from Figure 5 show a higher level of OS CPU resource usage than the other two workloads because the former one requires much more intra-cluster network communication and writing to disk.

The important conclusion is that for all three workloads we do not see any significant performance degradation when the web page set grows to about 1.4 million pages or 20GB of data in all tables, and we are using four cluster nodes each with about 128MB of RAM for file cache. Also in all workloads the CPU utilization percentage remains above 90% showing that the disk I/O time is largely overlapped with computation. This means that our technique works well at providing an efficient way to access and manipulate large sets of disk-resident data.

5.2.2 Comparison between Data-Driven and Control-Driven Approaches

The data in Figures 7 and 8 contrasts the performance characteristics of the hit list construction workload when we change the amount of memory from 256MB to 45MB and the number of partitions from 128 to 1 for each cluster node. Setting the number of partitions to one per node effectively converts the data-driven approach into a control-driven one. The difference from the pure control-driven approach is the small overhead of enqueuing and dequeuing of procedures' input data

[5] All data sets except the largest one are breadth-first crawls of `sunysb.edu` domain starting from `http://www.sunysb.edu`. The largest data sets is composed of a portion of pages referenced from ODP directory at `http://dmoz.org`.

All reported data points are averages over the four cluster nodes.

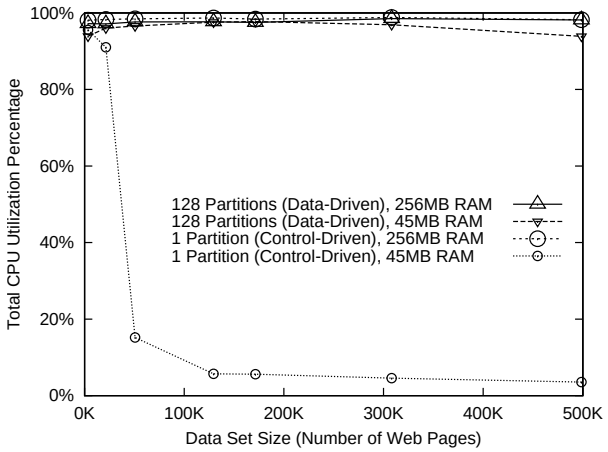

Figure 8: Total CPU utilization comparison between data- and control-driven approaches when constructing hit lists for different data set and RAM sizes.

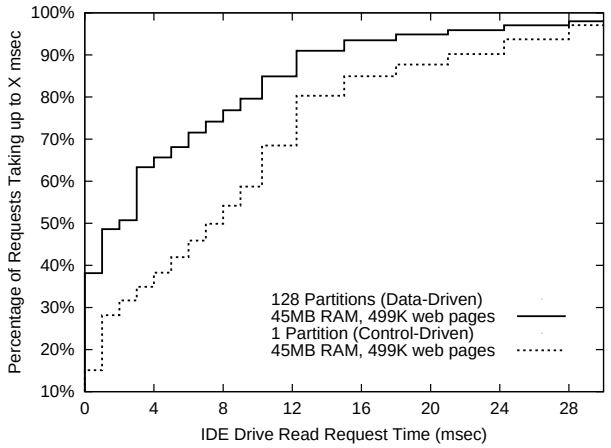

Figure 9: Disk reading latency comparison between data- and control-driven approaches when constructing hit lists.

and the separation of all processing in time into different workloads. Since the procedures' inputs are enqueued and dequeued in the same order as they are generated and there is no partitioning within each cluster node, the resulting data access pattern corresponds to the pure control-driven approach within each workload.

Figure 7 shows that reducing the amount of memory (thus making the data more disk-resident) does not noticeably affect the performance of the data-driven approach. Switching to the control-driven approach alone reduces the performance by a factor of 4.7 for 0.5M web pages. Since the CPU utilization percentage remains at 98% in this case, still not much random disk accesses are needed. Therefore, the reason for this performance drop is the increase in CPU cache misses due to larger working sets, which are though still mainly memory-resident. When we reduce the amount of RAM to 45M per node so that the working sets become mainly disk-resident when processing 0.5M web pages, the performance of the control-driven approach further drops by a factor of 22 to 34 procedures/sec due to a substantial number of non-sequential disk accesses issued, as evidenced by the fact that the CPU utilization drops to 3.5% in this case.

At one point the control-driven approach's performance is lowered to 20 procedures/sec when processing 0.5M web pages using 45MB of RAM. This is the theoretical minimum of performance for this workload, assuming that each procedure needs five 9-msec disk seeks. In contrast, the data-driven approach is able to sustain a 2800 procedures/sec performance rate when handling 1.4M web pages, which is within a factor of two of the peak performance rate of 5200 procedures/sec, when the entire data set is memory-resident. Thus, for the hit list construction workload, the data-driven approach's performance is at least around $4.7 * 22 = 103$ times better compared with the control-driven approach when the working set substantially exceeds the available system memory, and around 4.7 times better when the working set mostly fits into memory. One can obtain similar performance improvement results for other workloads (possibly under different data set or memory sizes), provided their working set size under the control-driven execution exceeds the amount of available memory.

5.2.3 Disk Utilization Efficiency

The other advantage of the proposed data-driven approach is that its disk access pattern is more sequential because the control flow then does not dictate the order in which data blocks are to be fetched from the disk. We have instrumented the IDE driver in the Linux kernel to directly evaluate the degree to which our data-driven technique avoids disk seeks when manipulating disk-resident data by measuring the time taken by individual read requests to the IDE disk. Figure 9 shows the histograms of the percentages of IDE disk read requests that took different amount of time to complete while performing the hit list construction task on 0.5M web pages over a 4-node cluster with 45MB of memory in each node. The data manipulated in this workload is about 730MB per cluster node, hence the processing is out-of-core.

The curve for the data-driven approach shows that a majority of disk reads take a smaller amount of time and thus are sequential. Although some disk seeks are required for disk reads in this case, in particular because the OS may write out dirty pages as the disk reads are on-going, the overall CPU utilization percentage is still very high, around 94%. The curve for the control-driven approach reflects, as expected, much less sequentiality than for the data-driven approach.

Figures 7 and 9 conclusively demonstrate that the

data-driven approach not only has a much better "reuse" factor for every disk block brought in, but also requires less overhead per disk I/O. As a result, the data-driven approach towards data preparation in large-scale search engines can perform two orders of magnitude better than the control-driven approach on modern hardware for out-of-core data sets.

5.2.4 Scalability of Clustering Implementation

There are two aspects of clustering effectiveness: how much are the communication and synchronization overheads and how well the work is balanced among the nodes. The data for URL score querying using our data-driven technique for the 1.4M document collection shows that for such uniform workload which is both communication-intensive and disk access-intensive our data-driven execution and communication model can achieve over 92% CPU utilization on every cluster node due to high parallelism and weak synchronization requirements: In this workload the fractions of idle CPU time for the four cluster nodes were 7.2%, 6.4%, 7.9%, and 4.1%; and around 1.3% was used by the other processes on each node. It is hard to single out cluster communication costs, but the amount of cluster traffic for each node is at most of the same order as its disk exchange traffic. Data partitioning using simple hash functions over object names is effective at providing partition sizes that are much less diverse than the sizes of individual data items in the partitions. As a result, the remaining unevenness of resulting processing on different cluster nodes leads to about 90% effectiveness of using a 4-node cluster versus 4-times faster execution on one node for the three different workloads when handling the 1.4M document collection.

6 Conclusion

Internet-scale search engines need to build various types of data structures to efficiently support a large number of user queries against a substantial portion of the global Web. Preparation of this data poses a unique engineering challenge because of the sheer volume of the external-memory data sets involved. The key architectural requirement for data manipulation in search engines thus is avoidance of random and/or small disk I/O. Fortunately the nature of search engine applications allows us to concentrate on improving the throughput of large groups of modifications, not the latency of individual updates.

In this paper, we described a data-driven technique to organize data manipulation that triggers computation only when its required data is brought into memory and amortizes the cost of disk I/O over large groups of computation operations. We have analyzed its applicability conditions, showing that it can be easily used whenever we have large streams of reads and/or updates that do not have to be executed immediately and which operate on partitionable external-memory data structures. Although conceptually simple, the technique is enormously powerful as it greatly improves the disk and memory data access locality and converts small/random disk I/Os into large/sequential ones, as well as increases CPU throughput via higher cache hit ratio. As a result, the efficiency of the data preparation process is significantly improved: Our experiments demonstrate that when the manipulated data sets grow far beyond the memory size, the execution performance improves by a factor around 100 for typical workloads compared with the straightforward control-driven execution model. In addition, our technique naturally combines with clustering to exploit the parallelism manifested in data manipulation. The experience of applying these techniques to all data manipulations tasks in a comprehensive search engine prototype *Yuntis* convinced us that these techniques can be equally effective in other data-intensive Internet applications with similar requirements.

To the best of our knowledge this is the first open-literature description of an I/O-driven external-memory manipulation technique of such universality and effectiveness, its application to data preparation in large-scale web search engines, and its detailed empirical performance comparison with the control-driven approach, which demonstrates its hundred-fold performance advantage on a functionally complete and fully operational search engine prototype *Yuntis*.

The future work includes selective use of asynchronous disk I/O when and only when doing so actually outperforms the use of non-blocking disk I/O with read-ahead by the OS. We also plan to investigate the tradeoffs of dynamic migration of processing and/or data among cluster nodes to achieve better overall cluster utilization.

7 Acknowledgments

This work was partially supported by NSF grants IRI-9711635 and MIP-9710622. The paper has also benefited from the comments of the anonymous reviewers.

References

[1] Sedat Akyurek and Kenneth Salem. Placing replicated data to reduce seek delays. In *Proceedings of the USENIX File Systems Workshop*, pages 133–134, Berkeley, California, May 1992. USENIX.

[2] AllTheWeb, `http://www.alltheweb.com/`.

[3] AltaVista, `http://www.altavista.com/`.

[4] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the Web. In *ACM Transactions on Internet Technology*, volume 1, pages 2–43. ACM Press, June 2001.

[5] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The Connectivity Server: fast access to linkage information on the Web. In *Proceedings of 7th International World Wide Web Conference*, 14–18 April 1998.

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of 7th International World Wide Web Conference*, 14–18 April 1998.

[7] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the Web: experiments and models. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[8] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. Technical report, Stanford University, California, 2001.

[9] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[10] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS '93)*, pages 714–723, Palo Alto, California, November 1993.

[11] Google, `http://www.google.com/`.

[12] Taher Haveliwala. Efficient computation of pagerank. Technical Report 1999-31, Stanford University, California, February 1999.

[13] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, December 1999.

[14] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[15] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, San Francisco, California, 25–27 January 1998.

[16] Maxim Lifantsev. Rank computation methods for Web documents. Technical Report TR-76, ECSL, Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York, November 1999.

[17] Maxim Lifantsev. Voting model for ranking Web pages. In Peter Graham and Muthucumaru Maheswaran, editors, *Proceedings of the International Conference on Internet Computing*, pages 143–148, Las Vegas, Nevada, June 2000.

[18] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the Web. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, May 2001.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the Web. Technical report, Stanford University, California, 1998.

[20] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Information Retrieval*, pages 105–112, Berkeley, California, August 1999.

[21] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, Berkeley, California, January 1990. USENIX.

[22] Raymie Stata, Krishna Bharat, and Farzin Maghoul. The term vector database: fast access to indexing terms for Web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[23] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[24] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, 20(3):225–242, March 1990.

[25] Chuan-Kai Yang and Tzi-Cker Chiueh. I/O-Conscious volume rendering. In D. Ebert, J. M. Favre, and R. Peikert, editors, *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym-01)*, pages 263–272, Wien, Austria, May 2001.

[26] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 243–258, Berkeley, California, October 2000. USENIX.