The VLDB Journal (2009) 18:739–764
DOI 10.1007/s00778-008-0121-2

REGULAR PAPER

Self-tuning management of update-intensive multidimensional data in clusters of workstations

Vassil Kriakov · George Kollios · Alex Delis

Received: 12 September 2007 / Revised: 8 August 2008 / Accepted: 28 October 2008 / Published online: 23 January 2009

Abstract Contemporary applications continuously modify large volumes of multidimensional data that must be accessed efficiently and, more importantly, must be updated in a timely manner. Single-server storage approaches are insufficient when managing such volumes of data, while the high frequency of data modification render classical indexing methods inefficient. To address these two problems we introduce a distributed storage manager for multidimensional data based on a Cluster-of-Workstations. The manager addresses the above challenges through a set of mechanisms that, through selective on-line data reorganization, collectively maintain a balanced load across a cluster of workstations. With the help of both a highly efficient and speedy self-tuning mechanism, based on a new data structure called *stat*-index, as well as a query aggregation and clustering algorithm, our storage manager attains short query response times even in the presence of massive modifications and highly skewed access patterns. Furthermore, we provide a data migration cost model used to determine the best data redistribution strategy. Through extensive experimentation with our prototype, we establish that our storage manager can sustain significant update rates with minimal overhead.

V. Kriakov
Polytechnic Institute of New York University,
Brooklyn, NY 11201, USA
e-mail: vassil@cis.poly.edu

G. Kollios
Boston University, Boston, MA 02215, USA
e-mail: gkollios@cs.bu.edu

A. Delis (✉)
University of Athens, 15771 Athens, Greece
e-mail: ad@di.uoa.gr

Keywords Multi-dimensional data · Cluster of workstations · Self-tuning storage

1 Introduction

Many modern applications continuously produce large volumes of data which require efficient mechanisms for storage, effortless growth, and speedy access methods [29,35,49,55,56,58,64]. For example, the Terra spacecraft (EOSDIS project [58]) and Landsat 7 [64] produce 150–200 GB/day of geophysical data. Through very high speed streams, such data are collected in analysis centers and have to be organized in a way that facilitates not only fast querying but also effective update handling, often entailing ever expanding storage requirements. Indexing methods that are capable of providing such efficient data access are needed to achieve these goals [41,42,48]. This efficiency must persist even when data access and update patterns vary over time due to continuously changing user interests, weather conditions, morphing traffic patterns and congestion points, sensor node failures, network topology records, or other causes specific to the application. Such unpredictable patterns often manifest themselves as hot-spots in distributed storage management systems.

In this paper, we address the aforementioned issues without resorting to specialized hardware. To support the high demands due to growing multidimensional data sets, the foundation of our proposal consist of a networked storage manager distributed over a cluster of workstations (COW) on a high speed LAN or switch. We achieve a cost-effective way of catering to ever-growing data sets by employing off-the-shelf workstations with plentiful disk space as well as powerful CPUs and system resources. Our core mechanism for providing efficient access to the multidimensional data is based on the R*-tree [2,14] which offers numerous benefits over similar indices [22,46,47,52]. More specifically, the R*-tree uses extensive optimization heuristics and forced-reinsertions as an alternative to splitting overfilled nodes [2]. A particular feature of R*-trees that our storage manager leverages is the ability to prune entire subtrees and insert them elsewhere in the index structure without disrupting the consistency of the underlying data [2]. This facilitates our self-administered data migration process for load redistribution without affecting the integrity of the data set. In our storage manager, the R*-tree structure is modified to a minimum in order to preserve its guarantees on utilization and performance.

We maintain that the process of migrating data from "hot" sites to less loaded ones must undergo careful consideration and cost-benefit analysis, especially in light of skewed access and update patterns. To identify such skews, a memory-efficient structure for access and update pattern analysis is necessary. To this end, we introduce the in-memory *stat*-index which provides finer granularity statistics for index nodes closer to the root and coarser granularity information for nodes at deeper levels. The *stat*-index is dynamically tunable allowing for adjustment of the trade-off between accuracy and memory consumption. In the process of selecting data for migration, the *stat*-index's real benefits stand out when data access and update patterns exhibit significant skews. In this context, our proposed storage manager employs the information provided by the *stat*-index in order to migrate a minimal subset of data, resulting in a very efficient load redistribution. The entire self-tuning process is devoid of external administrative assistance and, thus, is highly scalable for the continuously changing access skews due to fluctuating user demands.

A network coordinator maintains a global view of the load distribution among the sites. The coordinator's minimal function set allows it to scale up well with thousands of sites. This is accomplished through a distributed collaborative algorithm for self-identification of "hot" sites within the COW. Load rebalancing is considered only when a site deems itself to be overloaded. When this occurs, the overloaded site queries the coordinator for an under-utilized site that will ultimately receive data. The remainder of the protocol is carried out between the two sites which negotiate the nature and volume of data to be migrated. The dynamic and self-sufficient manner of this approach allows the COW to gracefully grow or shrink in accordance with the volume of data present. This is facilitated simply by attaching or removing sites from the LAN-based storage manager. In addition to efficient hot-spot dispersal, our proposed system consistently attains short response times while handling extensive and frequent data updates. This is aided by a query aggregation scheme which accumulates similar updates and queries for batch processing. To accomplish this without disrupting the integrity of the data, a locking mechanism for efficient R*-tree access is used, allowing queries and updates to propagate through the tree simultaneously [65].

These novel features of the COW-based manager make up our proposal and, along with a developed prototype and its evaluation, constitute our research contribution. Our system prototype is a full-fledged implementation written in C++/BSD-Sockets that runs on a network of Sun SPARC-stations and HP servers. Our main performance indicators are throughput, the system's load variance and the average response time (ART) of requests (queries or updates). ART provides a true scale for distinguishing performance differences as observed by the client [13].

We carry out extensive experiments to demonstrate the benefits of our proposed techniques. We use a very large synthetic multidimensional data set of 100 million data elements consuming up to 1 TByte of disk space and subject it to intense query and update transactions arriving at rates of 10 ms and updating over 30% of data set in a time frame of just a few minutes. The main results of our evaluation are:

1. During peak loads caused by skewed access patterns and frequent updates, our dynamic load balancing mechanism offers substantial improvements of a factor of three as compared to other self-tuning systems [27] and two orders of magnitude difference in performance compared to a non-self-tuning system [51].
2. Hot spots are dispersed through the system within very short periods of time, with minimal overheads attributed to the maintenance of the *stat*-index, even for data set sizes of hundreds of gigabytes.
3. Our COW manager exhibits robust scalability characteristics and provides for uninhibited growth with minimal human intervention.

To our knowledge, this is the first work to address the issue of dynamic load balancing in a COW with data sets experiencing high update rates and skewed access patterns.

The remainder of this paper is structured as follows: Sect. 2 discusses related work. Section 3 describes the architecture of our system and outlines the proposed load-sharing and data migration techniques, while the cost-models used in the data selection process are described in Sect. 4. Section 5 details our query aggregation optimizations and our experimental analysis is presented in Sect. 6. Conclusions and future research directions are discussed in Sect. 7.

2 Related work

Various distributed index structures for one-dimensional data have been proposed in the past. File-based indexes using distributed extendible and linear hashing (*LH**) where proposed in [9,32]. Extensions for support of multiple attributes in flat files was introduced in the *k-RP** [31]. An improvement over quad-trees was introduced as the *hQT** data structure, presented in [23], which adapts for 2-dimensional highly skewed file-storage. A shared-memory environment over parallel disks is explored in [57] where the problem of concurrent index processing is investigated. [33] also deals with indexing in shared-memory under multiple processors. In [18], the problem of scheduling query processes and background maintenance tasks on a distributed system is examined. The *B-link* tree [19] can support multiple levels of parallelism through a shared-nothing distributed approach, locking mechanisms, and partial data replication. A client-server approach for storage management is provided in [37].

A large body of research also deals with load balancing. The trade-offs between sender-initiated and receiver-initiated adaptive load sharing are discussed in [8]. An *LH** based load-conscious approach using a distributed random tree is proposed in [28]. This method provides for storage space utilization guarantees accomplished through data partitioning across a set of servers. Load balancing techniques for parallel disks are explored in [50], where "heat"-tracking is used to identify "hot" files which are ultimately striped and reallocated across a set of disks. The issue of on-line reorganization for centralized B^+-tree is investigated in [66], whereas methods of merging two B^+-tree that cover the same key ranges are discussed in [54]. Preservation of QoS guarantees even during data migration is discussed in [44]. A globally height-balanced adaptive parallel B^+-tree, termed AB^+-tree, for one dimensional data sets is introduced in [30] and its performance is examined through simulation. A "semi-distributed" version of the R-tree is proposed in [26] where formulae are provided to optimize leaf node sizes, called striping units, so that small queries are quickly retrieved from a single machine, while large queries engage as many sites as possible. [51] extends this concept to a shared-nothing R-tree architecture with each site indexing the local data set through its own R-tree, while the master site contains an R-tree of all non-leaf nodes with its leaves pointing to specific sites. [21] seeks a similar goal through parallelism exploitation by multiplexing the R-tree's leaves and internal nodes with cross-disk pointers in a multi-disk single-CPU system. Proximity index criteria are introduced for assigning new nodes to disks to optimize the probability that nodes spatially close to one another are stored on different disks. In [40], a method for parallel bulk loading of spatial data is presented. Furthermore, in [39] an algorithm to execute nearest neighbor queries on a parallel R-tree that is based on the method in [26] is discussed.

Load balancing in the context of multi-dimensional data is introduced in [34] where the R-tree is split between two tiers: one centralized, holding the root of the tree, and one distributed, indexing portions of the data set at dedicated client sites called processing elements (PEs). Every 10,000 queries the client PEs send load information to the master which then decides whether the system is balanced or if reorganization is necessary. The original R-tree structure [14] is relaxed to allow for as few as one element per node; in this way, dummy nodes are used to increase the height of subtrees pruned for migration. The system proposed in [34] is evaluated through a simulation study. In [16], we distribute the multi-version R-tree (MVR-tree) which indexes the history of spatio-temporal data across a set of sites; the most current version is held in the main memory of the server. This allows for fast lookups of the latest locations of objects, with complete archives of all historical data.

In this paper we introduce a number of techniques which substantially differentiate our effort from previous ones. Our main contribution is a COW-based storage manager which sustains very high update rates of multidimensional data in an adaptable and scalable manner. This is accomplished with the aid of our proposed *stat*-index. Our COW-based manager identifies "hot" sites within the cluster and with the help of the *stat*-index is also able to locate "hot" data within a site. We provide a cost-model for analyzing the potential benefits of data reorganization versus the expected overheads and, thus, allow for data migration to occur only under necessary and near-optimal conditions. Furthermore, we employ a buffering scheme for group execution of queries and updates so that only individual subtrees are locked during sequences of updates. Finally, in support of our contributions we provide extensive experimental results from a fully functional prototype system using very large synthetically generated data sets.

Some theoretical work has looked at providing polynomial time approximation algorithms for the NP-hard problem of determining the optimal migration plan from multiple sources to multiple destination data migration. The migration optimization techniques in [17] work under the assumption that there is a space constraint at the individual sites. Migration is performed in multiple stages using temporary sites to hold objects while migration is taking place. The goal of the paper is to perform migration in as few stages as possible where there is a correlation between the number of stages and the number of bypass sites necessary to complete the operation. In our work, we assume that there is always sufficient disk space in each site in the cluster. Furthermore, we perform migration in bulk, moving entire subtrees in order to reduce to time to re-insert objects in the index, whereas [17] operates on a per-object basis. In [24], replication is used at the file or block level under the assumption that data must be distributed amongst multiple attached disks. [24] is an extension to [17] in the sense that copy operations are added so that data can be replicated amongst the disks. This may work well for static environments, but in a highly dynamic setting where data is constantly modified, data replication becomes a much more difficult problem. Both [17] and [24] do not tackle the problem of which data is selected for migration. One of the core arguments of our paper is that this is a crucial decision which needs to be made in order to minimize the time to perform migration. Furthermore, the destination problem, i.e., deciding which disks/sites will receive data, is sidestepped in these papers by assuming that the destination disk(s) are known a priori. Using a Network Coordinator, we introduce a heuristic approach to identifying destination sites for data migration. Generally, using an approximation algorithm to calculate a migration plan in a distributed system requires a centralized view of the current state of the entire system and may become hard to scale up even with a small number of sites. The combination of utilizing one of these algorithms to construct a migration plan with our work on identifying the data to be migrated and the destination sites is certainly an interesting problem that, however, goes beyond the scope of this paper.

Preliminary results of the work presented here appear in [27] where we introduce a distributed collaborative algorithm for data migration based on the analysis of a variable level indexing scheme at the network coordinator. Our algorithm in [27] performs load sharing without the use of a *stat*-index and without the query aggregation techniques introduced here.

3 Fundamental system features

In this section, we describe the overall architecture layout of our manager, our query processing algorithm, and the interactions of the individual components necessary to facilitate our self-tuning mechanism. We also introduce our specialized data structure, called *stat*-index, and describe how it provides for an efficient data selection process.

3.1 COW-based storage manager architecture

Our model consists of a COW which communicate over a high-speed network and host the underlying data set. One of the sites in the cluster also serves as a load balancing coordinator. This site is aware of the workload at each workstation in the cluster and is referenced when an overloaded site requests to perform data migration to an underloaded site. The coordinator need not operate on a dedicated workstation as its resource consumptions are minimal, allowing it to quite easily run on a workstation that is also involved in the storage management. As seen in Fig. 1, each site maintains a portion of the entire data set, indexed locally by an R*-tree. A query can originate at any site. The originating site broadcasts the query to the other site and is responsible for collecting the result sets once they are available. The operations supported by the proposed storage manager are containment and intersection queries as well as insertions and deletions.

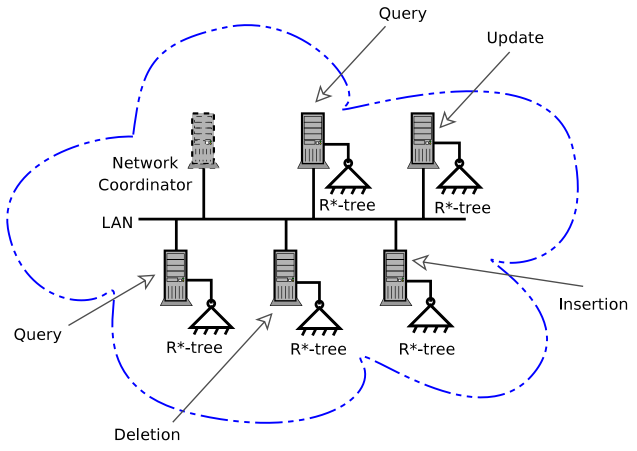


Fig. 1 Logical architecture: each site autonomously maintains its own R*-tree, while a Network Coordinator keeps an up to date table of workload levels for every site. To scale-up the system, additional workstations can be added seamlessly to the COW simply by attaching them to the LAN

To facilitate self-tuning (Sect. 3.3), each site maintains local throughput statistics used to determine whether it is overloaded. If a site determines that it is overloaded, an action is triggered to rectify the situation as outlined in Sect. 3.4. When data migration must take place, an overloaded site selects a fragment of its local data and migrates it to a less loaded site. The data selection process is based on access statistics maintained by each site as discussed in Sect. 3.5. The ultimate responsibility of any site at the receiving end is to take over the task of maintaining and providing fast access to migrated data. In the following sections, we examine the reasons for our design choices and discuss the interactions among the sites in the cluster.

3.2 Query processing

One of the main novelties of our storage manager is a completely decentralized query processing system. Past research efforts rely on a “distribution catalog” maintained by a dedicated workstation—a coordinator which is consulted for dispatching queries to the pertinent sites in the cluster [26,51]. Although this approach may be beneficial in a relatively static environment, it introduces two significant problems: first, when a large number of objects are inserted or updated, the index structure changes substantially, requiring the propagation of these modifications to the distribution catalog; second, the catalog becomes a bottleneck and hinders up-scaling as it is the central point of entry for all requests.

In [27], our experiments suggest that, under conditions involving heavy update loads, the best performance is achieved when a broadcasting approach is taken in order to deliver queries to the storage sites. The fundamental premise for this choice is that the benefit of a non-centralized indexing scheme outweighs the disadvantage of activating all sites on every query. In a highly dynamic environment, where the underlying data set is frequently updated, any form of centralized query processing will inhibit performance and, ultimately, will restrict scalability.

In this work, we define the "update" operation as a two-step action involving a delete request, immediately followed by an insert request. Insertions are performed in a decentralized fashion in order to cope with a potentially high rate of insertion requests. An insertion request (or a group of requests) can be submitted to any of the sites in the cluster. Using a pre-defined hash function known a priori by all sites in the cluster, insertions are shipped to potentially other sites in the cluster where the data is finally inserted in the local R*-tree. The hash function distributes the insertion requests uniformly across all sites in the cluster. When the number of sites in the cluster changes, the hash function changes as well. This is facilitated through the network coordinator which, upon the registration of a new site in the cluster, informs all other sites of the new hash function. Delete operations are broadcast to all sites in the cluster, in the same manner as queries.

The individual steps of our query processing workflow are illustrated in Fig. 2. When a query is (1) submitted to one of the sites in the cluster, this site (2) broadcasts the query on the network and becomes responsible for (3) collecting the results and (4) delivering the final set of results to the user. This method eliminates the problems that plague a centralized approach. Our Network Coordinator's functions are solely related to load balancing and are completely orthogonal to query processing. Therefore, in the context of query processing and data insertion/deletion, we claim that our architecture is completely decentralized.

3.3 Self-tuning principles

We introduce a number of effective heuristics to deal with dynamic self-tuning under heavy update rates of indexed multi-dimensional data. Dynamic load balancing is a necessary component in a COW environment which is subject to changing access patterns. In order for dynamic load balancing to be beneficial, however, we must ensure that it satisfies the following requirements:

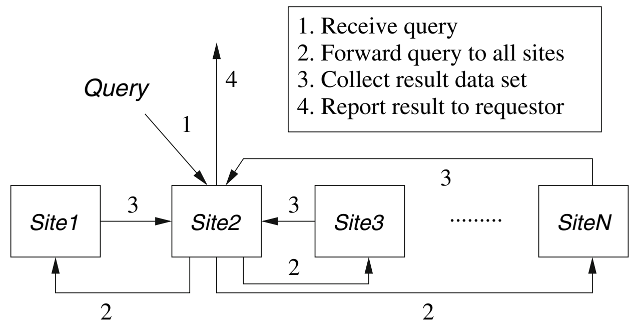

1. Receive query
2. Forward query to all sites
3. Collect result data set
4. Report result to requestor
Query
1
4
3
2
Site1
Site2
Site3
SiteN
....

Fig. 2 COW-based query processing: in our highly dynamic environment a completely decentralized query processing system ensures there are no bottlenecks such as the "distribution catalog" used in [26,51]

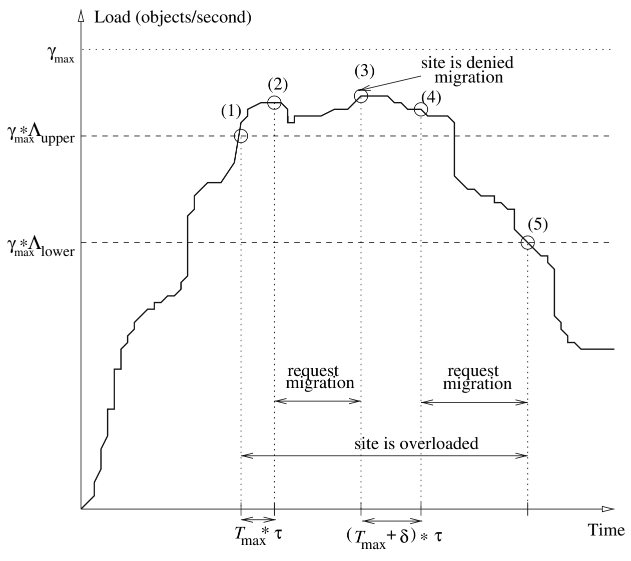


Fig. 3 Sample load variations for a given site

1. Load is shifted between sites without disruptions in the system's performance.
2. The overhead of self-tuning is more than compensated for by the resulting performance gains after completion of balancing.
3. No human intervention is necessary during the redistribution process.

In the following sections, we discuss in detail our methods used to satisfy these requirements.

3.4 Component interaction

We define "site load" (γ), or simply "load," as the number of objects retrieved per second by a site. In our experimental results (Sect. 6) this measure is also reported as throughput, and it indirectly corresponds to the CPU usage, disk I/O, and memory paging operations. We believe that defining the load in such a fashion truly reflects the working state of the system since the rate of data delivered depends directly and indirectly on these factors [13]. We will demonstrate the interactions of the various components of our COW-based storage manager with an example load distribution from one of the sites in the cluster as depicted in Fig. 3.

Each site i continuously measures its current load γ^i. This is depicted by the Load Monitor in Fig. 4. Sites have a fixed maximum sustainable load capacity γ^i_{max}, which can be

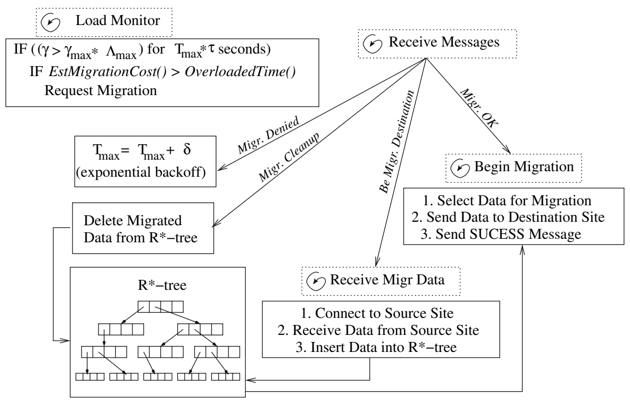

Load Monitor
IF ((γ > γmax* Λmax) for Tmax*τ seconds)
IF EstMigrationCost() > OverloadedTime()
Request Migration
Receive Messages
Migr. Denied
Migr. Cleanup
Be Migr. Destination
Migr. OK
Tmax = Tmax + δ
(exponential backoff)
Delete Migrated Data from R*–tree
Begin Migration
1. Select Data for Migration
2. Send Data to Destination Site
3. Send SUCESS Message
R*–tree
Receive Migr Data
1. Connect to Source Site
2. Receive Data from Source Site
3. Insert Data into R*–tree

Fig. 4 Overloaded sites request permission to perform migration. If the request is denied, $T_{\max}$ is incremented and no migration occurs. If the request is granted, data is carefully selected for migration, according to criteria detailed in Sect. 3.5, and is shipped to the recipient site, which in turn indexes the data through its own R*-tree

determined off-line by running a test benchmark [1] and is used to compute the current load percentage $\lambda^i = \gamma^i/\gamma^i_{\max}$. Sites use the two system-wide thresholds Λ_{upper} and Λ_{lower} to determine whether they are overloaded. Site i considers itself overloaded as long as the condition $(\gamma^i > \gamma^i_{\max} \times \Lambda_{\text{upper}})$ holds true for a specific period of time, or epoch [10]. These thresholds are represented by the horizontal dashed lines in Fig. 3, whereas the epochs are represented by the vertical dashed lines. The notion of epoch is used in order to stabilize the system under spurious jumps in a site's load. Furthermore, an exponential back-off algorithm [38] is adopted by increasing the duration of the epoch each time a migration request is denied. This improves system performance during extremely high loads as it reduces network congestion.

Prior to a site's first migration request, the epoch must last at least $T_{\max} \times \tau$ s, where τ is the interval of load measurements in seconds, and $T_{\max}$ is the number of load observations taken before migration is requested. This is visualized in Fig. 3 by the interval between points (1) and (2), where migration is requested at point (2). If the coordinator ascertains that the system is load balanced, it will reject requests for migration and the site will increase its $T_{\max}$ by δ for each rejected request. This occurs at point (3) in Fig. 3. $T_{\max}$ is reset to its original value when a migration request is granted. In addition, once a site is overloaded, it returns to normal state only when $(\gamma^i < \gamma^i_{\max} \times \Lambda_{\text{lower}})$. This event corresponds to point (5) in Fig. 3. While $T_{\max} \times \tau$ provides a temporal window to reduce the frequency of migration requests, $\gamma_{\max} \times \Lambda_{\text{lower}}$ and $\gamma_{\max} \times \Lambda_{\text{upper}}$ provide a load window to soften the entry to and exit from an overloaded state. The combined use of these windows provides for a stability in the migration initiation phase. The adjustment of the epoch as described above reduces the number of migration requests during system-wide overloads when self-tuning is not possible. Simultaneously, this mechanism provides short load balancing response times when skewed access patterns call for system tuning. In essence, the dynamic tuning of $T_{\max}$ allows sites to "learn" about the overall state of the system and to adjust accordingly. The parameters and measurements used to define the behavior of our self-tuning distributed storage manager are summarized in Table 1. We have performed experiments with a wide range of parameters and have determined that the numbers provided in Table 1 are indicative for a wide number of cases in our infrastructure. These thresholds serve as protective measures under extreme circumstances and small deviations in their values do not result in observable performance changes.

Reverting to the example in Fig. 3, the first point of interest (1) occurs when a site considers itself to be overloaded. After $T_{\max} \times \tau$ s, the site sends a Request Migration message to the network coordinator as indicated in Fig. 4. This triggers the self-tuning mechanism and the coordinator evaluates the system's state of balance as shown in Fig. 5. When overloaded sites request permission for migration from the network coordinator, the coordinator has to determine whether data migration would be beneficial, and if so, which site should receive the migrated data. The coordinator stores updates of each site's load percentage λ^i and its available disk capacity (dcap) in the Global Load Table (GLT) as indicated in Fig. 5. These updates are not periodic, but are piggy-backed onto other messages, eliminating frequent polling for load statistics on behalf of the coordinator and improving the scalability of our architecture. Sites which are performing data migration are noted in the GLT by recording the ID of the corresponding destination site (the site which receives the

Table 1 System parameters and representative values used in our experiments

Parameter	Symbol	Range of values
Maximum load capacity	$\gamma_{\max}$	3,000–30,000
Current load capacity	γ	30,00–30,000
Current load percentage	λ	$\gamma/\gamma_{\max}$ (0–100%)
Measurement period	τ	1 s
Max load measurements	$T_{\max}$	3–5
Load increment	δ	1–3
Upper load threshold	Λ_{upper}	50–85%
Lower load threshold	Λ_{lower}	40–75%
Load deviation	Δ	5–25%
Active sites	N	25–50
Dataset size	D	100,000,000

[1] To establish the value of $\gamma^i_{\max}$ for a particular site i, we issue a unit-square query, forcing the site to operate at its maximum sustainable load $\gamma^i_{\max}$. By issuing such a query we guarantee the leaves will be visited in arbitrary order, causing random disk accesses.

migrated data), in the dst column. For example, site 2 is in the process of migrating data to site 5. The coordinator stores timestamps (TS) of load updates in order to detect stale information and to explicitly request updates from sites that have not provided recent load information. This is a straightforward mechanism and is not discussed in detail here. With its global, yet minimal, view of the load distributions among the sites, the coordinator can decide whether the COW manager is balanced, as detailed below.

The load balancing mechanism is designed for speed and, thus, only considers the minimal amount of information kept in the global load table. The network coordinator computes the difference between the loads of the most ($\lambda_{\max}$) and least ($\lambda_{\min}$) loaded sites in $O(N)$ time, where N is the number of active sites. When the condition $(\lambda_{\max} - \lambda_{\min}) < \Delta$ qualifies, the system is balanced. The parameter Δ constitutes the system's imbalance relaxation factor and can be configured according to the specific application needs. We provide an empirical approach for determining Δ and experiment with values (5–25%) that are deemed representative for the parameter in order to evaluate its effect on the storage manager's behavior.

When the system is balanced, the coordinator dispatches a message to the requesting site declining the request for migration. If the system is deemed imbalanced, the least loaded site is chosen as the destination site and the requesting site is instructed to continue negotiations with that site. Since concurrent requests for migration may be issued by multiple sites, the coordinator marks current destination/source pairs in the GLT, indicated by the dst and src columns in Fig. 5. Such pairs of sites are not considered for destination candidates until the migration process between them completes. The following section discusses how the *stat*-index–a main-memory data structure used to track access skews in the R*-tree–can help to efficiently identify portions of the data set for migration.

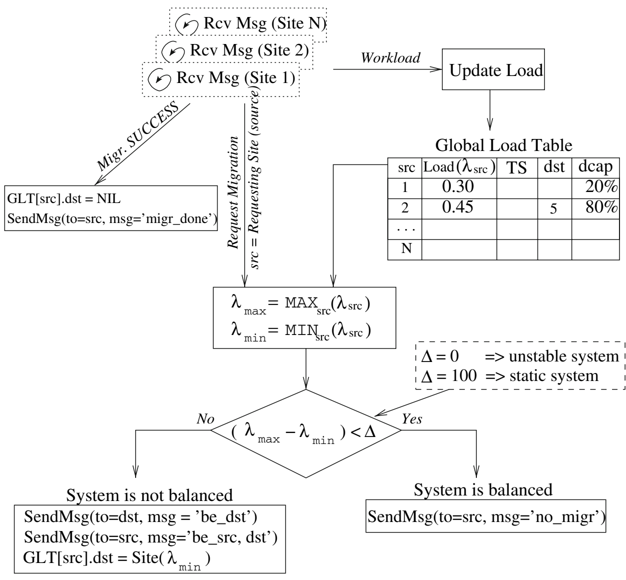


Fig. 5 The coordinator qualifies data migration based on the current load of each site, and selects for destination sites with the least load. Sites that do not have free disk space available (dcap) are not considered as destination sites

3.5 Data migration

3.5.1 Statistics-based data selection

We propose an efficient mechanism to collect access and update statistics that allow each site in the cluster to select a minimal amount of data for migration, while maximizing the effect on load redistribution. This reduces the overhead of data transfers among the sites and increases the system's self-tuning responsiveness. We demonstrate the advantages of the proposed mechanism through analysis and experimentation.

In the context of skewed access patterns following Zipfian distribution, we maintain that it is of significant importance what data is claimed for migration. If access patterns are uniformly distributed in space, in order to achieve a desired load reduction, say 50%, a site has to migrate an equivalent portion (50%) of data. However, when access patterns are skewed, the degree of skew determines the amount of data to be migrated. For example, a workload where 90% of the accesses are targeted at 9% of the data, only 5% of the "hot" data must be migrated to achieve a 50% load reduction. Figure 6 visualizes the relationship between load reduction rates and data migration size for various types of access skew. As shown by the graph, in order to achieve a desired load reduction, very few elements must be redistributed under higher skews as compared to a uniform access distribution. Thus, by exploiting access pattern information, migration overheads can be reduced substantially.

In order to be able to identify the skews of such patterns, we monitor and record detailed statistics on data accesses and updates. Maintaining such information can be quite costly in terms of space and processing time. Ideally, statistics for each node in the R*-tree should be collected. However, maintaining this information is prohibitive since each R*-tree access, regardless of whether it is a read or a write, would incurs the cost of a write when an access counter is updated. For this reason, previous efforts have elected to maintain minimal information only at the root level [34]. We show through analysis and experimentation that this simplification is insufficient as it does not provide us with the information necessary to determine which data to migrate, especially in environments with highly skewed access patterns.

Traditional histogram-based approaches are inadequate for tracking such statistics because of shortcomings in their update process. Methods for incremental updates of multidimensional histograms were proposed in [12], but they rely on

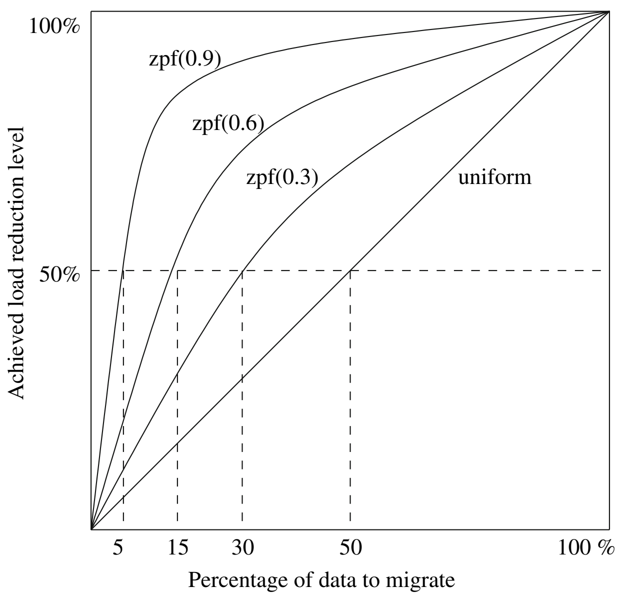


Fig. 6 With skewed access patterns, the amount of data that must be migrated to achieve a desired load reduction sharply decreases as compared to uniform access distributions. Therefore, it is important to identify skewed access patterns when dealing with data redistribution

random samples of the underlying data. On the other hand, *STHoles* [4] are built by analyzing the results of queries and [60] builds the histograms using sketches. However, our goal is to monitor access to all nodes at different levels of the R*-tree structure and, therefore, none of these approaches apply. Hence, we propose a main-memory structure which is a form of a hierarchical histogram and is targeted specifically for optimization of the R*-tree migration process. The *stat*-index is built on top of the R*-tree, as depicted in Fig. 7. The *stat*-index tracks the read/write counts for nodes and groups of nodes in the R*-tree. The structure is very efficient and automatically adjusts the quality of statistics depending on the amount of main memory that it is allocated.

The *stat*-index can be viewed as a downward-thinning tree layered atop the R*-tree. Each *stat*-index node contains the $read_count$s and $write_count$s for node(s) in the R*-tree, a timestamp indicating when these counters were reset, and a child pointer ($\langle timestamp, read\_count, write\_count, child\_ptr\rangle$). To limit the number of nodes in the *stat*-index, we use a partial mapping from the R*-tree index nodes to a reduced space index. We consider a single R*-tree node to be an f-dimensional vector, where f is the tree fanout, and map it to a ϕ-dimensional vector, which is a single node of the *stat*-index with $\phi < f$. We propose the mapping function $m(i) = \lceil i \times \phi/f \rceil$, which translates groups of f/ϕ elements in an R*-tree node to one element in a *stat*-index node. With this node dimensionality reduction, the growth rate of the *stat*-index is a constant factor lower than that of the R*-tree. This is not sufficient to guarantee that the *stat*-index will fit in main memory. For example, with $\phi = f/2$ the *stat*-index has nodes with fanout half of the actual R*-tree's, and if we assume $f = 100$, then $\phi = 50$ and for a fully utilized tree of height 5, the number of nodes in the *stat*-index is over 300 million. To further control the amount of main-memory space used by the *stat*-index, we provide a scheme where the subtrees of the *stat*-index become "thinner" at deeper levels in the index, as seen in Fig. 7. In this manner, the growth rate of the *stat*-index can be bound to a decaying function such as e^{-h}, where h is a node's distance from the root. To accomplish this, we alter the mapping function so that it takes into account h: $m(i, h) = \lceil i \times \phi(h)/f \rceil$ where the function $\phi(h)$ provides a variable fanout which depends on the node's distance from the root. One instance of $\phi(h)$ which results in the *stat*-index's subtrees growing at an increasingly lower rate is $\phi(h) = f/2^h$. This provides finer granularity statistics for nodes closer to the root at the cost of coarser statistics at deeper levels in the index.

For a limited memory size, we can adjust the fanout function to accommodate the *stat*-index at the trade-off of reduced statistics detail at lower levels in the tree. Furthermore, the height of the *stat*-index is bounded as its deepest level can contain only one element per node. For example, the *stat*-index depicted in Fig. 7a cannot grow further after reaching a height of 3. This limitation is a direct result of the function

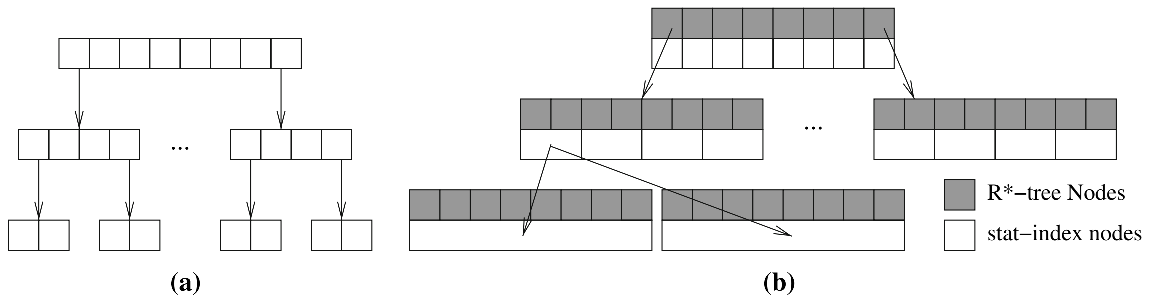


Fig. 7 **a** Complete structure of a *stat*-index with $\phi(h) = f/2^h$, **b** and an overlay of some of the R*-tree nodes with the *stat*-index. At lower levels in the trees, statistics are maintained for fewer elements per node. Each *stat*-index node consists of the tuple $\langle timestamp, read\_count, write\_count, child\_ptr\rangle$

$\phi(h)$ and the original tree's fanout f. For $\phi(h) = f/2^h$, the maximum height of the *stat*-index is $\lfloor \log_2(f) \rfloor$. For example, the *stat*-index in Fig. 7a cannot grow deeper than $\log_2(8) = 3$ levels.

The total space requirements in terms of elements for a *stat*-index of height H and variable fanout $\phi(h) = f \times a^h$ (where $a < 1$) is

$$\text{Total Elements}^{\text{stat-index}}(H) = \sum_{k=0}^{H} f^{k+1} a^{\frac{1}{2}k(k+1)} \quad (1)$$

See Appendix A for a derivation. In fact, the number of elements in an R*-tree is a specific instance of Eq. 1 where $\phi(h) = f$ and

$$\text{Total Elements}^{\text{R*-tree}}(H) = \sum_{k=0}^{H} f^{k+1} \quad (2)$$

We maintain that such a reduction guarantees that the *stat*-index is sufficiently small to fit in main memory. To illustrate this, consider an R*-tree with $f = 128$ and $H = 4$. Under 70% utilization, the R*-tree can index over 180 million objects and, with typically 64 bytes per element, it consumes roughly 1.4 TB, while its *stat*-index with $\phi(d) = f/3^d$ and 32 bytes per element, consisting of the tuple $\langle timestamp, read\_count, write\_count, child\_ptr \rangle$, consumes about 30MB.

The worthiness of the *stat*-index is explored in Fig. 8 where the access patterns of an exponentially distributed 2-dimensional query workload at a single site are displayed in gray-scale gradient, with darker areas corresponding to more frequently accessed data. White areas correspond to data which is migrated away from the site. Figure 8 depicts the results of two experiments: in the first instance, data is chosen for migration based on the *stat*-index, and in the second, data is arbitrarily selected. Figure 8a displays the access patterns of the workload at a site when no migration is performed: this is the baseline case for comparison. In Fig. 8b we see that the migrated data is from the region of high activity. Thus, the workload skew at the site is reduced and the query encompasses a wider range of data at reduced access rates as seen in Fig. 8b. When the storage manager has no information about the access skews of the data that it manages, it arbitrarily selects data for migration as can be observed in Fig. 8c. This visualization substantiates our claim that sufficiently detailed access statistics can introduce considerable improvements in the quality of the data selection algorithm.

3.5.2 Concurrency control policy

Simultaneous queries, insertions and updates through an R-tree are possible as described in [25], where the R-link tree was introduced. Our choice of R*-tree, renders those methods unusable because of one main difference: re-insertions. Therefore, we only focus on concurrency control in the context of data migration [45]. Guaranteeing strict ACID properties is beyond the scope of this paper and has been shown to be an impediment [43] for contemporary applications that support spatio-temporal data management. However, our concurrency control policy does guarantee the consistency of the data migrated between sites during load balancing. The issue at hand is important not only from a synchronization standpoint but also because of the overheads involved during data reorganization. Thus, we allow query processing and data migration to occur simultaneously. To facilitate this, the subtree chosen for migration is marked prior to commencing data migration. Queries are allowed to enter this subtree at no cost. When updates enter a marked subtree, they are propagated to the destination site which received the migrated data as shown in Fig. 9.

Once migration commences, two copies of the data may exist: one at the destination site and one at the source site.

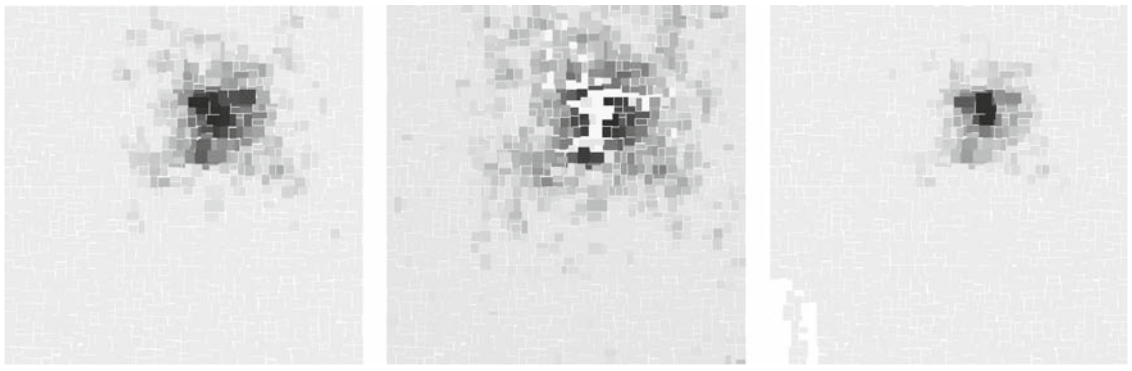

(a) Access Patterns **(b)** Using the Stat-Index **(c)** Arbitrary Data Selection

Fig. 8 Visualizing the effect of using the *stat*-index to migrate "hot" data: darker areas designate "hotter" regions. In white are the areas where data is not present because it has been migrated to a different site

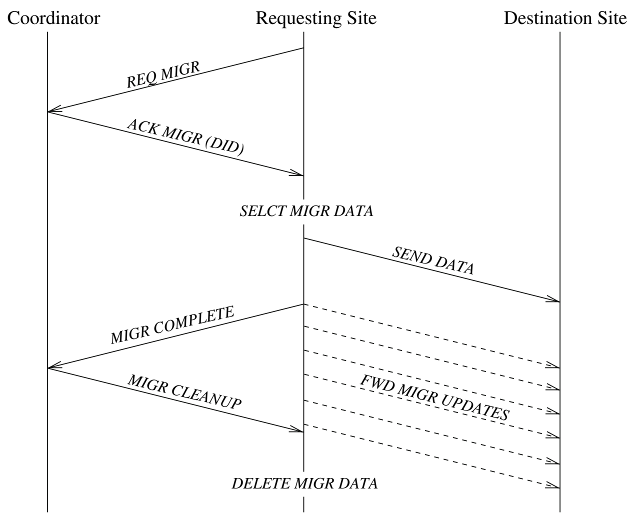


Fig. 9 Concurrency control

The forwarding of updates to the destination site guarantees the consistency of the migrated data since the modifications to the subtree being migrated will be carried over at the destination site. When migration is completed, the requesting site sends a "migration complete" message to the coordinator. The coordinator immediately sends a "migration cleanup" message to the requesting site. When the requesting site receives this message, it safely deletes the migrated data and stops forwarding messages to the destination site. At this stage, the data migration process is complete and there is only one up-to-date copy of the migrated data at the destination site.

3.6 Storage manager optimizations

The process of data selection and migration may consume large number of CPU cycles and I/O operations affecting the normal query processing functions of the COW-based storage manager. Although, the *stat*-index allows for quick and efficient data selection, data migration presents a more significant operational overhead. During the migration process, the data must be fetched from disk, packed and transmitted to the destination host. This can be I/O, CPU and possibly network intensive. Previous experiments revealed high spikes in the mean query response time during data migration periods [27]. This attests to the resource monopolization by the self-tuning mechanisms. To alleviate this additional load from a host that is already overloaded, we propose three optimization mechanisms: (1) subtree locking, (2) reinsertion forwarding, and (3) query aggregation.

3.6.1 Subtree locking

To ensure that data migration and query processing can proceed simultaneously, we lock the subtrees which have been selected for migration so that they cannot be modified. Any updates that get blocked by such a migration lock are forwarded on to the destination site along with the migrated data. This allows the data migration to proceed concurrently with query processing while ensuring consistency in the presence of updates. Queries are allowed to enter the locked subtree, as long as the subtree is not pruned due to a "migration cleanup" message from the network coordinator, as described in Sect. 3.5.2. Insertions are not allowed to enter a locked subtree since it will eventually be pruned. Instead, insertions are forwarded to the underloaded destination site, where they are handled in a straight-forward manner by inserting them in the local R*-tree. Deletions that affect elements in the locked subtree which is being migrated to the remote destination site are forwarded to the destination site and are applied to this subtree before it is inserted in the local R*-tree.

3.6.2 Reinsertion forwarding

During normal workloads, each site handles re-insertions due to local R*-tree node splits. This increases the site's workload, but the added benefit is improved R*-tree quality. Reinserting all nodes in a tree can improve quality by up to 30% [2]. If the local site is overloaded and is in the process of performing data migration, the added cost of reinsertion can cause further increase in load. In such a case, we piggy-back the reinserted items on the data messages sent to the destination site. Locally, this alleviates the costs of R*-tree restructuring. The destination host is selected by the coordinator due to its low workload, therefore, it is a good candidate for the data to be reinserted. Furthermore, a channel of communication (TCP) already exists between the source and destination sites, so no additional resources (sockets, memory, etc.) are required. At the destination host, any piggy-backed reinsertion items are handled as regular insertion requests—each one is inserted individually at the top of the R*-tree.

3.6.3 Query aggregation algorithms

In a system which performs dynamic load balancing, the overhead of selecting and migrating data can create significant delays in the normal execution of query processing [27]. We propose efficient query aggregation algorithms that alleviate the problems caused by this overhead. The idea is to cluster similar queries that may have accumulated during a data migration phase and to execute them in bulk in order to reduce response time as compared to executing them one at a time. The details of our proposed query aggregation algorithm are presented in Sect. 5. When sufficient number of insertion requests are accumulated on the queue during data migration periods, the insertions are bulk-loaded into the local R*-tree, significantly improving response time as compared to one-at-a-time insertions. This effect is later analyzed in Sect. 6.4 where we report that bulk-loading can improve throughput by up to 23%.

3.7 Scalability

Due to the data migration policies described above, our COW manager provides a flexible environment for automatic system up-scaling with the growth of the data set. New sites can be introduced into the distributed system by attaching them to the same network cluster. The coordinator immediately detects the new site and establishes a communication link. This situation presents an unbalanced system and leads to data migration from the most loaded site(s) in the system to the newly added site. Gradually, part of the data set is relocated to the new site, alleviating the workload on the other sites.

Down-scaling of the system is also accomplished in a similar manner: the site being taken down signals the coordinator that it needs to shift all of its data out to other sites. The network coordinator directs the site to be removed to the least loaded destination site. If the two sites can negotiate that all data be migrated, the down-scaling concludes after the data is shifted. Otherwise, as much data as possible is shifted and a new request is sent to the coordinator. The process proceeds until all data is shifted and the site can be taken down safely.

4 A cost model for data selection

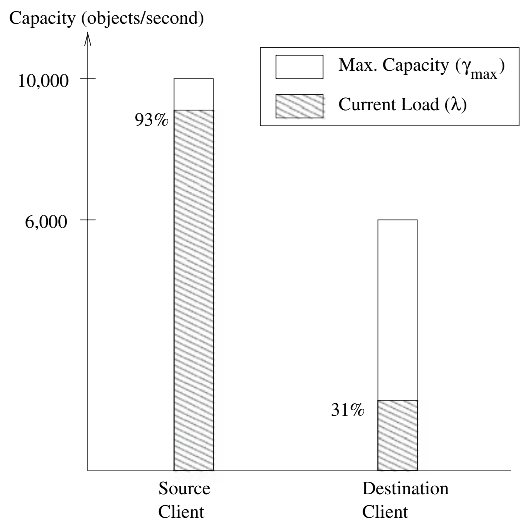
In order to estimate the overhead due to our data selection and migration techniques, we develop an analytical cost model. Each site uses this model to determine whether the cost of performing data migration will be outweighed by the expected benefits of load redistribution. This action is performed by the LoadMonitor at each site as shown in Fig. 4.

4.1 Costs involved

In order to understand and evaluate the costs of data migration, we take a detailed look at the intricacies of the data selection process. We assume that the system is in an imbalanced state, a site has requested permission for migration, the coordinator has selected a destination site, and the two sites have established a communication link. First, the source site must determine a target load rate λ_{target}, which it will attempt to attain through migration of appropriate portions of the "hot" data. Essentially, λ_{target} is the equilibrium load between the destination and source sites: $\lambda_{\text{target}} = \lambda_{\text{src}} + (\lambda_{\text{src}} - \lambda'_{\text{dst}})/2$, where the actual destination site's load λ_{dst} is normalized using the source and destination site's maximum load capacities: $\lambda'_{\text{dst}} = \lambda_{\text{dst}} \times (\gamma^{\text{dst}}_{\text{max}}/\gamma^{\text{src}}_{\text{max}})$. This normalization is necessary in order to account for heterogeneous sites of different capacities γ_{max}. Figure 10 shows a specific example of a source site which can deliver up to 10,000 elements per second and is 93% utilized, and a destination site which supports up to 6,000 elements per second and is 31% utilized. Had the two sties been homogeneous, the source site would have had a target load rate of $93 - (93 - 31)/2 = 62\%$. However, due to the destination site's lower capacity, the actual target load rate of the source site is 72%.

With λ_{target} computed, the source site selects R*-tree nodes whose load accounts for $(\lambda - \lambda_{\text{target}})$ percent of the total R*-tree load. The total R*-tree load is computed from the loads of each subtree st off the *stat*-index root:

$$\text{Load(root)} = \sum_{\text{st} \,\epsilon\, \text{root}} \frac{(\alpha \times \text{st.reads} + \beta \times \text{st.writes})}{(\text{time\_now} - \text{st. timestamp})} \qquad (3)$$



γ^{dst}_{max}	6,000
γ^{src}_{max}	10,000
λ_{dst}	31%
λ_{src}	93%
λ'_{dst}	31 * 10,000 / 6,000 = 52%
λ_{target}	93 - (93 - 52) / 2 = 72%

Fig. 10 Example state of heterogeneous source and destination sites during data migration. The target load rate is computed from the normalized loads of the source and destination sites

where α and β are weight coefficients for adjusting the significance of reads relative to writes since writes are usually more costly. Load(root) represents the frequency of reads and writes applied to the entire R*-tree. The data selection process traverses the *stat*-index, selecting a set of nodes $\mathcal{S}$ from the R*-tree, so that:

$$\sum_{n \in \mathcal{S}} \text{Load}(n) = \text{Load}(\text{root}) \times (\lambda - \lambda_{\text{target}}) \tag{4}$$

This is accomplished through Algorithm 1, with the initialization *SelectData*(root, Load(root) $\times(\lambda - \lambda_{\text{target}})$). The algorithm performs multiple data migrations for each subtree that is selected for migration, allowing any buffered queries to be processed between *MigrateSubtree*() operations. Furthermore, using information from the *stat*-index, the algorithm considers the size of the subtree selected for migration in order to estimate the expected time to perform migration. If this time is deemed large relative to the longest migration that has taken place so far (if any), each child of this subtree is migrated individually, again allowing query buffering to take place between migrations. In this algorithm, the function

Algorithm 1 $SelectData(Node : n, TargetLoad : \lambda_t)$

```
1: if Load($n$) $< \lambda_t$ then
2:   if Subtree n is not large then
3:     $MigrateSubtree(n)$
4:   else
5:     for each child $c$ of $n$ do
6:       $MigrateSubtree(c)$
7:     end for
8:   end if
9:   return $Load(n)$
10: else
11:   $curload \leftarrow 0$
12:   $\{S\} \leftarrow SortByLoad(n.children, 'desc')$
13:   for $s_i \in \{S\}$ do
14:     $curload \leftarrow curload + SelectData(s_i, \lambda_t - curload)$
15:     if $curload >= \lambda_t$ then
16:       return $curload$
17:     end if
18:   end for
19:   return $curload$
20: end if
```

Load(n) returns the corresponding load as recorded by the access information in the *stat*-index. The algorithm traverses the R*-tree in a depth-first fashion, following first the most loaded subtrees.

If a subtree's load is not sufficient to reach the target load reduction (i.e., if $\text{Load}(s_i)/\text{Load}(\text{root}(\text{R}^*)) < \lambda - \lambda_{\text{target}}$), the subtree is selected for migration and the algorithm terminates. Otherwise, the children of that subtree are examined recursively. If the leaf-level is reached and no subtree is selected for migration, the node in the leaf with the highest load is selected. Since the *stat*-index's height is at most as much as the R*-tree's height H, the data selection process runs in time $O(H \times \phi(h))$. At each level the process analyzes at most $\phi(h)$ elements to find the maximum loaded one. In the worst case $\phi(h) = f$, and the running time is $O(H \times f)$, where $H = \log_f(D_i)$ and D_i is the local data set size at site i. Therefore, the asymptotic time complexity for *SelectDataCost* is $O(H \times f)$.

The *stat*-index employs a hash look-up method based on the mapping function $\phi(h)$. In our experiments we use $\phi(h) = f/2^h$. The *stat*-index hashes R*-tree nodes to its nodes using page number and node height. For nodes where $\phi(h) < f$ the *stat*-index adjusts the read/write counters by the corresponding $\phi(h)/f$ ratio to normalize the statistics for nodes with coarse-grained counters.

Now we can estimate the cost to select data for migration and to transmit it to a destination site in terms of time (seconds):

$$\begin{aligned}
&\text{MigrationCost}(i) \\
&\quad = \text{SelectDataCost} \\
&\qquad + \text{NwkRate} \times M_i \times \text{ObjSize} + \log_f(N_i) \times M_i \\
&\qquad + \log_f(N_d) \times M_i
\end{aligned} \tag{5}$$

where the first factor is the cost to select data for migration as previously discussed. The second factor is for transmitting the data over the network where *NwkRate* is the rate of transmission (in bytes/s) and M_i is the amount of data (in bytes) selected for migration from site i. *ObjSize* is the size of the objects (in bytes) being potentially moved. The last two factors are for one-at-a-time deletion from the source site with data set cardinality N_i, and one-at-a-time insertion at the destination site with cardinality N_d, respectively. These two factors can be reduced to $\log_f(N_i)$ and $\log_f(N_d)$ if the operations are done in bulk [1,3,5,6,11,62,63]. The amount of data to migrate M_i depends on the degree of access skew at site i, the desired target load reduction level λ_{target}, and the cardinality of the data set N_i managed by the source site i. When performing bulk deletions and insertions, the cost of migration depends on the amount of data selected for migration M_i. As we have postulated, by leveraging information of the local access skews, Algorithm 1 allows us to reduce M_i so as to achieve a more even load redistribution, while reducing the cost of performing load balancing.

To validate the effectiveness of our *stat*-index, we look at the opportunity for optimization in a Zipfian skew access model.[2] In Fig. 11 we plot M_i as a percentage of K for three different λ_{target}. As expected, Fig. 11 shows that significant performance improvement can be acquired when data accesses are skewed. For example, at $z = 0.7$, the cost of migration

[2] For this purpose we take $N_i = K$ objects and assign access frequencies such that item k has access frequency of $f(k) = 1/k^z$ where z is the Zipf factor, which we vary from 0.1 to 1.0, with lower values denoting less skew.

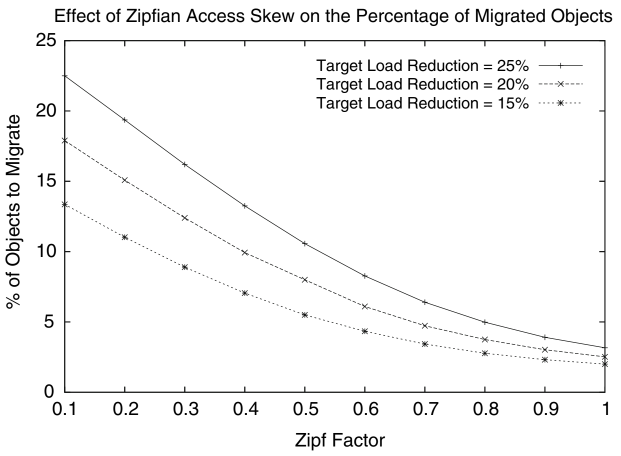


Fig. 11 Zipfian access distribution model

can be reduced four times as compared to a self-tuning system which assumes uniform access distribution. We use this analytical model to build a table of the portion of data expected to be migrated for various skew factors and for different target load reduction rates. When data must be migrated, the source site uses this table to estimate M_i. The skew factor is calculated from a second-order (non-linear) regression on the load information at the leaves of the *stat*-index, and the target load reduction rate is calculated as previously explained. With M_i at hand, and all other parameters of the Migration-Cost(i) formula known, the source site can estimate its total cost of migration in units of seconds.

To determine whether or not it is worthwhile to perform data migration, the source site compares the amount of time it has spent in overloaded state versus the estimated cost of migration. When the former exceeds the latter, we expect data migration to alleviate the load of the site and the site sends a migration request message to the Coordinator. Note that this decision is made independently at each individual site. This improves scalability and works under the assumption that another site exists which is sufficiently underloaded to allow the current site to reach its target load state. If this assumption is wrong, the Coordinator will simply deny the migration request.

4.2 Access statistics: cost of maintenance

There is some computational overhead involved with maintaining variable-granularity access statistics on nodes in the R*-tree. However, due to our hash-based implementation of the *stat*-index, it is possible to update access information in constant time, simply by pre-computing the dimensionality reduction matrix for a given mapping function. However, R*-tree insertions and updates are more difficult to deal with. For brevity and without loss of generality, we only discuss insertions. The methods for keeping the *stat*-index up to date with deletions or updates are analogous. In this scenario, due to the algorithms governing the R*-tree, three different scenarios are possible.

In the simplest case, a new object is inserted and no internal nodes in the R*-tree are modified. In this situation, the corresponding *stat*-index node of each R*-tree node visited has its read/write counter updated. With the hash mapping, this can be done in constant time.

The second case is when internal or leaf-level nodes split. If the new node maps to the same *stat*-index node as the original R*-tree node, no additional work is needed and we proceed as in the first case. If, however, the new node's corresponding *stat*-index entry is different, we must properly account for the shifting of the access statistics from one *stat*-index node to another by decrementing the access counters of the original *stat*-index node. It may not always be possible to determine the exact amount by which to decrement the old *stat*-index nodes. This information may be estimated from children nodes with variable accuracy, depending on the depth of the split node. For simplicity and speed considerations, it is best to assume a uniform split and redistribute the access information evenly between the original and the new *stat*-index nodes. If the assumptions is incorrect, it will only have a temporary effect since the correct skew statistics will propagate through the newly formed subtrees as new access information is collected.

The R*-tree differs from other R-tree variants in its introduction of a reinsertion strategy. The third scenario occurs during a reinsertion, where instead of splitting a node, a subset of its subtrees is chosen for reinsertion—the nodes are deleted from the current parent node and inserted at the top of the tree.[3] Instead of computing the appropriate adjustment for the corresponding *stat*-index node after the nodes are deleted, we simply reset the parent node's counter and update its timestamp. The rest of the deleted nodes are handled as in the other insertion cases. This strategy works well in our case because reinsertions would require us to identify as set of R*-tree nodes based on the *stat*-index node which is being affected. Since our *stat*-index mapping function in essence performs dimensionality reduction, a reverse mapping from a *stat*-index node to a set of R*-tree nodes would lead to loss of information. Thus, counter resetting helps to simplify the *stat*-index maintenance when R*-tree data is being deleted. After *stat*-index nodes are reset, they may not contain statistically significant information for some amount of time. During this time, the data selection algorithm relies on the information present in the parent node. The period during which the information in a reset *stat*-index node is not used is the current value of $T_{\max} \times \tau$ as explained in Sect. 3.4. This allows us to take a lazy approach for *stat*-index updates:

[3] The reasoning behind this is that reinsertions improve tree quality.

only one R*-tree node's *stat*-index node is updated during deletions, avoiding the updates of child *stat*-index nodes.

In all of the above cases, the cost of updating the *stat*-index information is constant. This concurs with our goal of keeping the *stat*-index small and efficient. The only conceivable practical limitations are due to a small overhead of accessing main memory.

5 Query aggregation algorithms

When a site's resources are consumed by the processing of data migration or very large queries, incoming queries are staged on the queue at that site. Let's assume that m queries are placed in the queue $\{Q_1, \ldots, Q_m\}$. There are a number of ways to handle this batch of queries. The simplest is to process each query one by one in a first-in-first-out (FIFO) order. This is inefficient if some of the queries request the same data, in which case the same pages may be read from disk multiple times. Another way to process the queries is to combine them into a single large MBR composed of their union [7]. This allows a single search through the local tree to be performed to answer all the queries on the queue. After the results are fetched, they are once again filtered through the individual queries. This method may also be inefficient if the union of the original queries covers a large area, retrieving a very large result set. In this case, if the available buffer space is not sufficient to contain the entire result set, thrashing will occur and, again, the same pages will be fetched from disk multiple times. Since there are no guarantees on the query workload and request may arrive from many different sites, it is very likely that the enqueued queries will cover a large area. We note that there has been some work [1] on executing multiple spatial queries in batch in order to amortize the cost of each I/O for multiple queries. However, this execution does happen on-line and is based on the idea of buffering groups of queries at different levels of the R-tree and moving queries to deeper levels when the buffer gets full [1]. Therefore, the response time of an individual query can be arbitrary large. Furthermore, the approach is based on a centralized R-tree and cannot be applied to our environment.

We propose a different way to efficiently process the queries from the queue. We combine the queries into a few groups (or clusters) of bulk queries and each group search is performed independently. We distinguish between three stages of this operation: (1) query clustering, (2) cluster-searching and (3) filtering. Our effort is focused on creating a good clustering algorithm to accomplish the first phase. Cluster-searching and filtering are straight-forward operations performed as explained above. The query clustering problem is the most important as it is responsible for creating "good" query-groups which will reduce the number of I/Os as compared to processing the queries individually.

Ideally, the coverage of each cluster should be carefully determined according to the distribution of elements in the local R*-tree and the query distribution. The larger the area of a cluster is, the more likely it is to contain results irrelevant to the queries in that cluster. However, the smaller the cluster area is, the more clusters will be formed and, in the worst case, there will be only one query per cluster. This trade-off is difficult to analyze on-line since it depends on the workload distribution, the data distribution, and main-memory availability. The time to analyze all these factors may outweigh the benefits. Therefore, our goal is to first decide whether query aggregation should be performed, or whether each query should be processed individually. For the purpose, given m queries, we consider three factors: the amount of query overlap A_o, the total query area A_q, and the total number of locally indexed elements N. The exact query overlap can be computed in $O(m^2)$ time simply by comparing each query with all other queries and keeping a sum of the overlapping areas. Below we show how we can give a good estimate for A_o in $O(m\log(m))$ time. Assuming that the data is uniformly distributed, the expected number of elements retrieved due to the overlapping areas is $A_o \times N$ (remember that elements are indexed in the unit square). In the presence of skewed data distributions, an approximation based on uniformity assumption still suffices as such an approximation is relative to the total number of elements in the tree. The expected number of elements retrieved by all the queries is $A_q \times N$. Assuming that each element retrieved from overlapping areas is reused at least once, the worst-case gain from performing query aggregation is:

$$g = 1 - \frac{(A_q \times N - \frac{1}{2} A_o \times N)}{A_q \times N} = \frac{A_o}{2A_q} \tag{6}$$

The gain varies from 0 when there is no overlap to 1/2 when there is complete overlap. For a cluster of two overlapping queries, the factor is exactly 1/2 and $A_o = A_q$. We use g to determine if aggregation is beneficial.

We evaluated empirically the overheads of the clustering algorithms proposed below and used the results to set a threshold which triggers the aggregation algorithm. As a result, if cluster-searching can save us at least 20% of disk I/Os ($\frac{A_o}{2A_q} > 0.20$) as compared to individual query execution, we utilize the query aggregation algorithm.

Once we have decided to proceed with query clustering, we must form "good" clusters in order to reduce dead space as much as possible. Dead space is the area covered by the MBR of the cluster, but not covered by any individual query in that cluster. If a large amount of dead space is created by the clustering algorithm, many irrelevant elements will be fetched during the cluster-search process, and will have to be eliminated during the filtering process. This causes I/O inefficiencies. We propose four methods for processing buffered queries.

5.1 K-means clustering for disk I/O optimization (K-DISK)

Although we will further see how our *stat*-index can be used for efficient query aggregation, our first approach utilizes a generic K-means clustering algorithm [20]. This algorithm produces K groups of queries such that for each subset of queries in each group some objective function is satisfied. The algorithm requires an initial set of K samples which are used to minimize the objective function. For each query, the objective function is evaluated for each of the K samples and the query is assigned to the sample group which minimizes the objective function. A second round is repeated with the K original samples replaced by the centroids of the newly formed groups. This process is repeated until there is no change in the centroids of the clusters. The efficacy and speed of this process are highly dependent on the quality of the objective function. As a basis for comparison, we use a distance function based on the p-norm Minkowski metric for n-dimensional vector space $\Re^n$:

$$d_p(x_i, x_j) = \left(\sum_{k=1}^{n} |x_{i,k} - x_{j,k}|^p\right)^{\frac{1}{p}} \tag{7}$$

Note that when $p = 2$, d_p is simply the 2-dimensional Euclidean distance. In the above formula, x_i and x_j represent the geometric centroids of the two queries which are being compared. The Minkowski metric will produce groups of queries that are close to one another in Euclidean space. Although, intuitively this should result in good overall clusters, for our purpose, we build an objective function which estimates the number of I/Os that each cluster will result in. Minimizing this function will produce clusters that cause the least number of I/Os.

The total number of disk access for each query q can be estimated as given in [61], for uniformly distributed data sets:

$$DA(q) = 1 + \sum_{j=1}^{1+\lceil \log_f \frac{N}{f} \rceil} \left\{ \frac{N}{f^j} \prod_{i=1}^{n} \left(\left(D_j \frac{f^j}{N} \right)^{1/n} + q_i \right) \right\} \tag{8}$$

where D_j is the density of nodes at level j, N is the total number of elements in the tree, f^j is the R*-tree fanout at level j and n is the number of dimensions.

5.2 K-means clustering with *stat*-index (K-SI)

As our main concern is with highly skewed data-sets, we perform the following modifications. We add a third attribute to each entry in the *stat*-index–the number of elements indexed by that *stat*-index node. This is similar to [59] where the *aP*-tree is specifically designed to answer range aggregate queries. This counter can be updated in a straight-forward manner during insertions and deletions. As previously shown, each insertion or deletion will go through the *stat*-index in order to update the appropriate read/write counters. Therefore, the only cost associated with this modification to the data structure is that it will have a slightly larger footprint in main-memory due to the increased size of each node. Armed with the *stat*-index element counters, we can calculate the density of elements under each group of R*-tree nodes enclosed by that *stat*-index node s_j as $D \times \frac{s_j^{<C>}}{N}$, where $s_j^{<C>}$ is the elements counter. To use this additional information in Eq. 8, we must perform the outer summation not for each level, but for each *stat*-index node:

$$\mathrm{DA}(q) = 1 + \sum_{j=1}^{\sigma} \left\{ s_j^{<R>} \prod_{i=1}^{n} \left(\left(D \frac{s_j^{<C>}}{N s_j^{<R>}} \right)^{1/n} + q_i \right) \right\} \tag{9}$$

where $\sigma = \sum_{k=0}^{H} f^{k+1} a^{\frac{1}{2}k(k+1)}$ as in Eq. 1 is the number of nodes in the *stat*-index with fanout $\phi(h) = f \times a^h$, and $s_j^{<R>}$ is the number of R*-tree nodes contained by *stat*-index node s_j.

The objective function that we propose exploits the above estimates of the number of disk access for a query $\mathrm{DA}(q)$ and for the intersection of two queries $\mathrm{DA}(u, v)$. We select the initial set of K clusters from the m queries on the queue which are furthest apart, as measured from their geometric centroids. The clustering algorithm is utilized only if at least $2K$ queries have aggregated on the queue. Otherwise, each query is performed separately, without any grouping. Given two queries u and v, the objective function we propose is:

$$\mathrm{d}(u, v) = \mathrm{DA}(u) + \mathrm{DA}(v) - \mathrm{DA}(u \cap v) \tag{10}$$

The K-SI clustering algorithm will group queries which result in clusters that require the minimum number of disk accesses. The net cost for a cluster of m queries can be simplified to:

$$\begin{aligned} \mathrm{DA}(\text{cluster}) &= \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} \mathrm{d}(q_i, q_j) \\ &= \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} \{\mathrm{DA}(q_i) + \mathrm{DA}(q_j) - \mathrm{DA}(q_i \cap q_j)\} \end{aligned} \tag{11}$$

Intuitively, the optimization function above attempts to maximize the query overlaps (intersections) in order to minimize the overall cost.

We further improve our approach by taking into account the underlying page caching mechanisms. Assuming that the DBMS utilizes a LRU (least frequently used) scheme to remove items from a main-memory buffer, we can use the *stat*-index to give bias to those overlapping queries which pull data from "hot" areas. To accomplish this, we assign a weight $w(u, v)$ to each pair of intersecting queries u and v. The weight is determined by querying the *stat*-index (SI) with the MBR of the two queries' intersection MBR$(u \cap v)$, summing up the read/write counts for the intersecting *stat*-index leaf nodes, and finally adjusting by the global (root-level) read/writes:

$$w(u, v) = \frac{1}{\text{reads (SI)} + \text{writes (SI)}} \times \sum_{s \in \text{Query (SI, MBR}(u \cap v))}^{n} (\text{reads}(s) + \text{writes}(s)) \tag{12}$$

One problem with K-means clustering is that K is an input to the algorithm. That is, we must have a priori knowledge of the number of clusters that we wish to form. To estimate a value of K we use the above-mentioned rationale that a good cluster is not very large, nor very small. To obtain a rough approximation, which is exact for the case when the queries are uniformly distributed in space, we use $K = (1 - \frac{A_o}{A_q}) \times m$. The more overlap there is among queries, the fewer clusters will be formed. At the extreme, if there is no overlap among the m queries, then m clusters will be formed, meaning that each query will be executed separately.

5.3 R*-tree Clustering (R*-CLUST)

In addition to the K-means clustering algorithm we investigate another approach for grouping the queries that have accumulated during the migration process. Here we use a hierarchical clustering approach. First, we build a temporary in-memory R*-tree of the MBRs of the queries. During this process, the MBRs of each query are grouped according to the criteria of the R*-tree insertion algorithm. These criteria aim at reducing "dead space" and non-leaf node overlap, which coincide with our clustering goals. Once the temporary tree is built, each group of queries is identified by selecting the set of internal nodes at depth d. We call each of these sets a grouping-subtree since they represent a subtree of the original temporary tree. Thus, the problem is reduced to discovering at what depth to select a grouping-subtree. The closer to the root the grouping-subtree is, the larger its area will be. The closer to the leaves it is, the more groups will be formed. Intuitively, this approach will result in good grouping because during insertion of objects the R*-tree algorithm forms subtrees such that overlap of non-leaf nodes is minimized. In order to solve the above trade-off problem, we overlap the in-memory R*-tree with the original R*-tree. The grouping-subtrees should encompass areas containing only as many elements as we can process in-memory.

The temporary R*-tree of the queries' MBRs can also be used to estimate A_o—the query overlap. After the tree is built, we need only compare sibling leaves. On average, there are $\log(m)$ siblings resulting in overall cost of $O(m\log(m))$, where m is the number of queries in the buffer. This will give an approximate answer since leaves in different subtrees may also overlap.

5.4 Depth-first search (AGGR)

We compare the above three approaches to a depth-first search through the R*-tree where all buffered queries are compared to each node retrieved from the R*-tree. This method, called AGGR, achieves the best I/O performance but can consume many CPU cycles since each R*-tree node that is read from disk must be compared to all queries in the buffer. Furthermore, the AGGR method might result in higher query response times compared to our clustering methods since all buffered queries complete at the same time, regardless of the size and spatial location of the query.

The four methods for grouping queries accumulated during the migration process are experimentally evaluated in the following section. For each method, we analyze the number of I/Os and CPU time spent in processing the aggregated queries relative to a baseline system which process them one at a time.

6 Experimental results

6.1 **Experimental setup and methodology**

To evaluate empirically the performance of the proposed system, we developed a working prototype in C++, which is distributed on a cluster of 50 Sun workstations interconnected through a 100 Mbps Ethernet network. Each workstation runs on a 400 MHz UltraSPARC CPU with 128 MB of main memory. Furthermore, we perform very large data set experiments of up to 1 TB of data on a cluster of up to 40 HP DL360 servers. We built our *stat*-index structure on top of an existing C++ implementation of the R*-tree [15]. Our primary objective was to compare the *quality* and *speed* of load-balancing of our *stat*-index based self-tuning system to a baseline system without a *stat*-index. In addition, we analyzed the performance gain resulting from the use of the query aggregation which leverages off of information maintained by the *stat*-index. We take a statistical approach to measuring the quality of load-balancing. Intuitively, a well balanced multi-site system is one which exhibits low fluctuations in the load distribution among the sites. Quantitatively, we measure this balance by recording the standard deviation (σ) of the loads among all of the participating sites. A low standard deviation means that the distribution of the loads across the sites is close to the mean (average) load of the entire system. For example, assuming that the loads are normally distributed across the sites, a standard deviation of 8% would mean that 68% of the sites' loads are within 8% of the mean and 95% are within 16% of the mean.

The speed of load balancing (ν) is defined as the time it takes the system to reach a steady state. Steady state is reached when the measured load distribution obtains the characteristics of a stationary process in stochastic terms. This means that the standard deviation (i.e., second-order statistics) is constant over time. In practice, a live system with consistent workload will experience small perturbations in the individual sites' loads. These small fluctuations mean that σ is variant with time and its rate of change will never become constant. Therefore, to establish a range for acceptable stationarity, we looked at the distribution of σ over time in a balanced system and measured the average of σ over 100 experiments. In Fig. 12, we see that when σ becomes stationary, it varies by $\pm 0.50\%$. For our experiments, we set the range to $\pm 1.0\%$ to allow for higher fluctuations. This value allows us to terminate the experiments when this steady-state is reached.

The baseline system which we use for comparisons supports self-tuning through data reorganization, but does not maintain any information on access skews. More specifically, the differentiating factor of the baseline system is the lack of the *stat*-index which we introduce in this paper. This baseline system resembles those previously discussed in [27,30,51]. In [27], we have shown that such a self-tuning system, without a *stat*-index, works well for a wide range of workloads. The baseline system lacks both the *stat*-index and the query aggregation algorithms introduced here.

For the purpose of testing the system's performance under high deletion/insertion rates, we focus primarily on a mixture of containment queries, as well as deletion/insertion operations, with the latter two comprising up to 30% of the workload. We use synthetically generated data due to difficulties in finding a sizable real-life[4] data-set for these experiments. The synthetic data set consists of 100,000,000 hyper-rectangles in unit space with mean sides of 0.0002 units and extensions of 0.00002 units. Thus, the expected density of the data set is 5 units. The generated objects are of size 40 bytes each, except for the very large data set experiment, discussed in Sect. 6.6, where we increase the object size up to 10 KBytes, creating a 1 TByte data set. We partition the data equally in space according to the number of sites and assign each fragment to a site. This method of partitioning allows us to directly control the workload skew in the system by specifying the density and spatial location of the queries. This is accomplished by generating queries spatially distributed according to either a normal ($N(\mu, \sigma)$) or an exponential ($E(\alpha)$) distribution. The area of each query is determined by drawing its length and width from a normal distribution with mean 0.01 units and standard deviation of 0.001 units. The queries are submitted every 10 ms from a randomly chosen site, which collects all the results and records the response time for each query. The response time is measured from the instant the query is submitted until all results are returned from all sites involved in answering the query.

Each experiment is repeated with data dimensionality varied from 2 to 4. For brevity, we report only the results for the 4-dimensional data. In general, data of higher dimensions consumes more space, which affects the system's performance as data reorganization require larger volumes of data to be migrated between sites [36].

One of the significant improvements of our COW storage manager is that it is tailored for handling very high deletion/insertion rates and skewed query workloads. To ascertain a sustained performance level under such a workload, during each experiment run we inject into the system insertion and deletion of data points, which account for a 10–30% change of the entire dataset; we perform up to 30 million delete and insert operations. Due to our query aggregation algorithms, our distributed storage manager can sustain a significantly higher deletion/insertion rate than an equivalent system without such buffering. As a measure of load we use the number of elements retrieved per second at each site and report that as a percentage of the site's maximum capacity. The maximum load capacity of each site is measured once for all experiments and is dependent on the workstation's hardware characteristics. As explained earlier, this allows us to run the system on a heterogeneous set of sites.

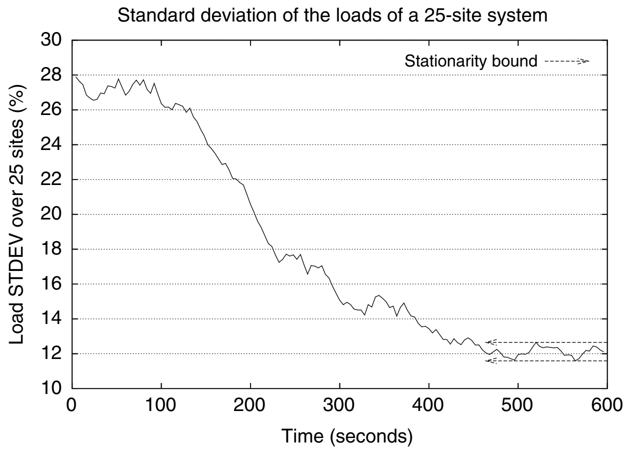


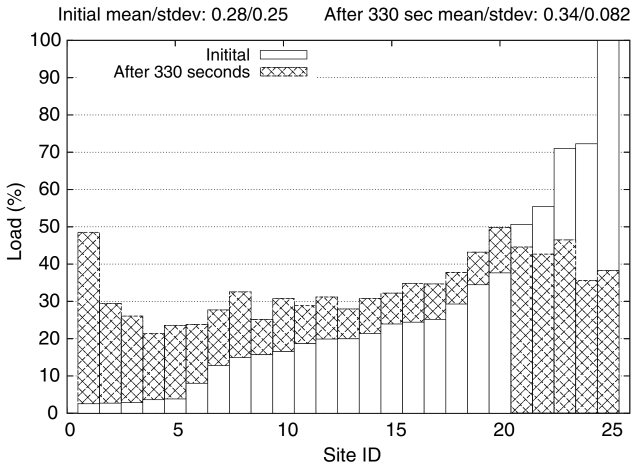
Fig. 12 Due to load-balancing the standard deviation of the loads of 25 sites is reduced to 12%. The fluctuation in the last segment, when stationarity is reached, is $\pm 0.5\%$

[4] http://www.rtreeportal.org, http://www.cs.du.edu~leut/MultiDimData.html.

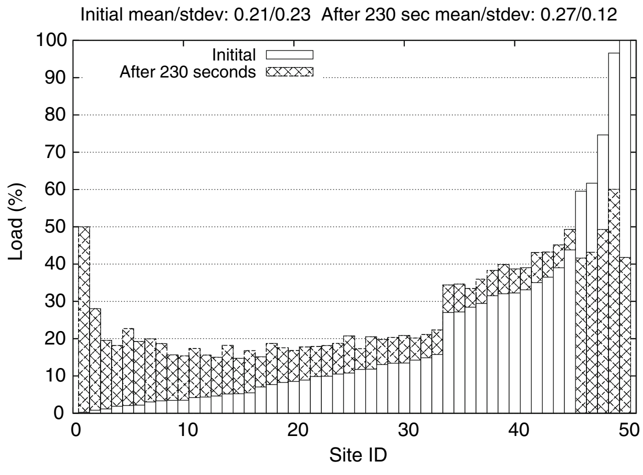
6.2 Efficient load distribution and scalability

We evaluated the quality and speed of load distributions under two different workloads with 25 and 50 workstations. In Fig. 13a and b we see the measured load distributions for a workload generated from an exponential distribution $E(\alpha = 0.20)$ for 25 and 50 workstations, respectively. The second workload is produced from a normal distribution $N(\mu = 0.5, \sigma = 0.15)$; the measured values for 25 and 50 workstations can be seen in Fig. 14a and b, respectively.

We ran the experiments until the standard deviation in the load across the sites remained within the stationarity limits for a period of 1 min. The results show that in the case of the exponential distribution, where there is more skew in the initial load, the system takes half the time to reach a steady state as compared to a normally distributed workload: Fig. 13a versus 14a. Under exponential distribution, the steady state is not necessarily a balanced state, because our threshold values prohibit sites from requesting migration if their load is below $\Lambda_{\text{lower}} = 65\%$, as per Table 1. This is the reason why the load distribution in Fig. 13b is skewed, even though a steady state is reached. Furthermore, if we compare Fig. 13a and b, we see that with more sites available in the system, the load-balanced state is reached faster. In this case the network coordinator has more choice in selecting underloaded sites for data migration. In addition, with more

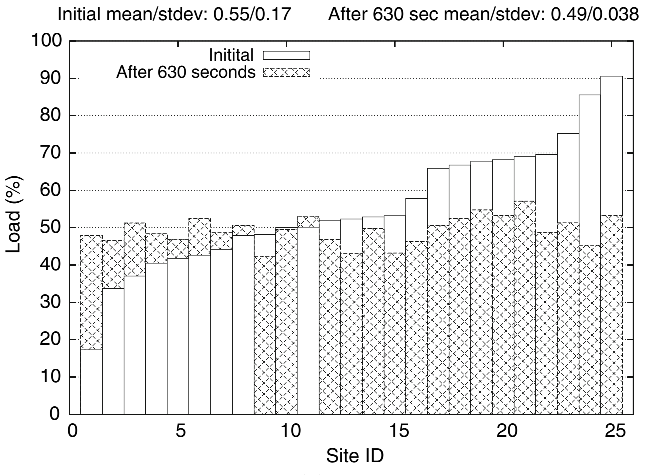


(a) 25 sites

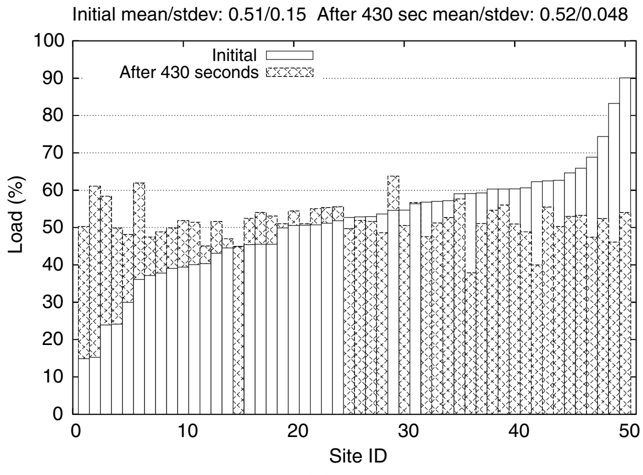


(b) 50 sites

Fig. 13 The workload is generated from an exponential distribution over 25 and 50 sites. The white bars show each site's load before migration and the shaded bars show each site's load after the system reaches a steady state. To simplify the visualization, the smaller of the two bars is always drawn "in front" of the larger one

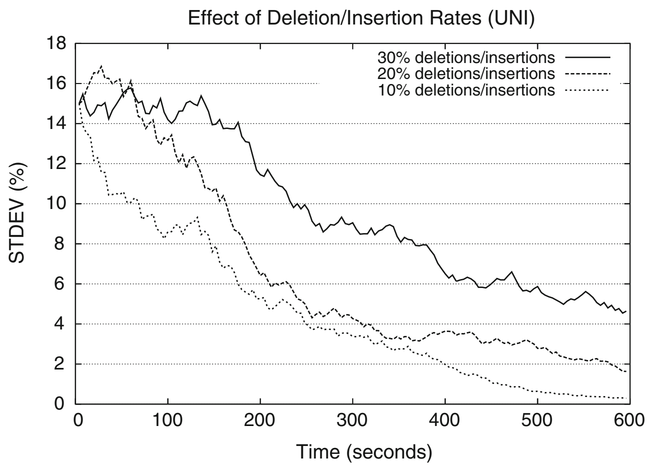


(a) 25 sites

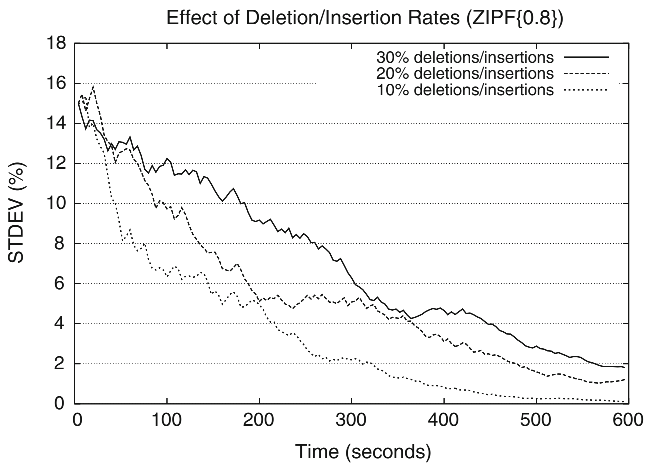


(b) 50 sites

Fig. 14 The workload is generated from a normal distribution over 25 and 50 sites. The *white bars* show each site's load before migration and the *shaded bars* show each site's load after the system reaches a steady state. To simplify the visualization, the smaller of the two bars is always drawn "in front" of the larger one



(a) Uniformly distributed workloads



(b) Zipfian distribution of workloads

Fig. 15 The coupling of the *stat*-index with a query aggregation algorithm allow the system to sustain very high deletion/insertion rates without significant degradation in performance. This performance overhead is further explored in Fig. 17

sites available, there is a larger probability that a sufficiently underloaded site will be found to receive enough data to offset the the load of an overloaded site.

The experiments with the normally distributed load have more interesting steady-state levels as is evident in Fig. 14a and b. The workload is controlled so that over time the average load on the system is just above 50%. However, the initial standard deviation is set to 15%. The sites are ordered by increasing load along the x-axis to visualize this better. Over time, the system is able to reduce the load variance significantly, with most sites reaching a variance within 3.8–4.8% of the mean as shown in Fig. 14.

6.3 Load distribution under high update rates

To evaluate the system's performance under different deletion/insertion rates, we measure the standard deviation of the load distribution through time. Our goal is to analyze how the load of the more overloaded sites changes through time for three different deletion/insertion rates (10, 20, 30%), under workloads with spatial distribution of the objects that is uniform (UNI) or Zipfian (ZIPF) with $z = 0.8$. As in previous experiments, the initial system state consists of a mean load of 50% and $\sigma = 15\%$.

As Fig. 15 shows, the deletion/insertion rates can have a significant impact on the system's ability to efficiently tune itself: it can take up to two times longer to reach the same state of balance when 30% of the data set is updated as compared to when only 10% of it is updated. Our self-tuning system provides significant improvements, and even when 30% of the data set is modified, it is capable of reducing the load by 15% within a time-frame of 2 min. Compare this to a baseline system which does not utilize a *stat*-index as depicted in Fig. 16. The two experiments in Fig. 16 are performed with

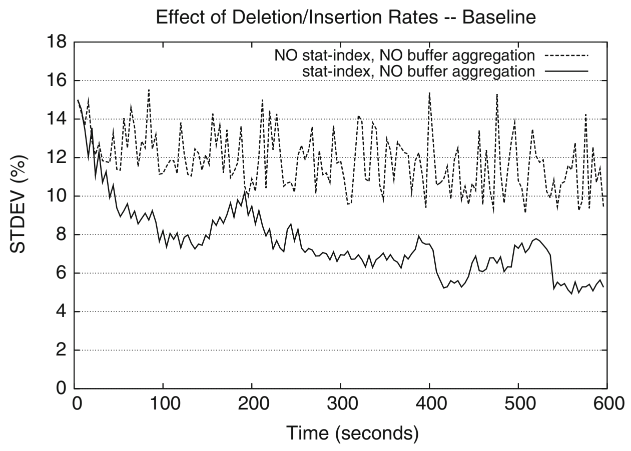


Fig. 16 Perfrmance of a baseline system (BASE) without *stat*-index and without query aggregation where 10% of the data set was updated

only 10% of deletion/insertion applied to the system. With higher deletion/insertion rates the baseline system (BASE), which does not use a *stat*-index and query aggregation, is not able to cope with the workload. In the baseline case, shown by the dashed line, the data used for migration is selected arbitrarily in incremental batches of 1,000 objects and query processing is suspended during periods of data migration. The behavior of this system is unstable, which is due to lack of an appropriate data selection policy. The bottom line on the chart reveals what happens when the *stat*-index is used to select "hot" data for migration while the query aggregation algorithm is switched off. While this is a significant improvement over the baseline system, it is still an inferior proposition compared to the results in Fig. 15. The intermittent "spikes" occur when spatial pockets of high update concentrations cause excessive write-locks of nodes in the R*-tree.

An interesting observation from Fig. 15 is that the rate of load reduction is often higher during the initial phase of the experiments. This occurs because the system first balances the most overloaded sites, of which there are few due to the initial load skew. However, this effect is not as significant under higher deletion/insertion rates. Under lower deletion/insertion rates, the initial data migration in the beginning of the experiment is performed very efficiently with little contention for resources. This is reflected by the very steep drop of the 10% line in Fig. 15a and b during the initial 100 s of the experiment as compared to the 20 and 30% deletion/insertion rate lines.

We have concentrated on exploring the effect of the *stat*-index on load redistribution over time, but the query response time is equally important as it gives us the user's point of view. Therefore, we measure the ART per query per second. The response time is the total turn-around time from the instance the query is submitted until all the results are returned to the user. With the workloads designed as previously described, the steady state response time is 1.35 s. This is depicted in Fig. 17. We see that under normal conditions, the overhead of using the *stat*-index is about 8%. We believe that this overhead is justifiable given the benefits gained during periods of highly skewed activity. Furthermore, this value can be controlled by adjusting the granularity level at which detailed access statistics are kept. This of course would be a trade-off at the expense of less efficient load balancing when high workloads occur.

The aspects of this trade-off can be further observed in Fig. 18 where we examine how a system without a *stat*-index performs load balancing compared to one that uses the *stat*-index: it takes almost ten times as long to reach a balanced state. In addition, during the initial period when a lot of data is migrated, the overhead of the baseline system is two to three times higher. This is expected because, without knowledge of the access patterns, much more data has to be reorganized to achieve an equivalent state of balance compared to a system with a *stat*-index. The response time is affected mostly during the period of data migration between sites since this is the time when most of the CPU, I/O, and network resources are allocated to the data migration process. The duration of this period is a direct function of the number of elements that need to be migrated as described in Eq. 5. The computational time for data selection is negligible when a *stat*-index is not used because data is selected arbitrarily. When the *stat*-index is used, the computation time is on the order of the height of the tree. Thus, these experiments concur with our earlier analysis of the model.

To further understand the benefits of the *stat*-index, we monitor the total number of elements that are migrated each second. The results are displayed in Fig. 19. As expected, the case of not using a *stat*-index does not improve noticeably over time. However, with our storage manager, the system is

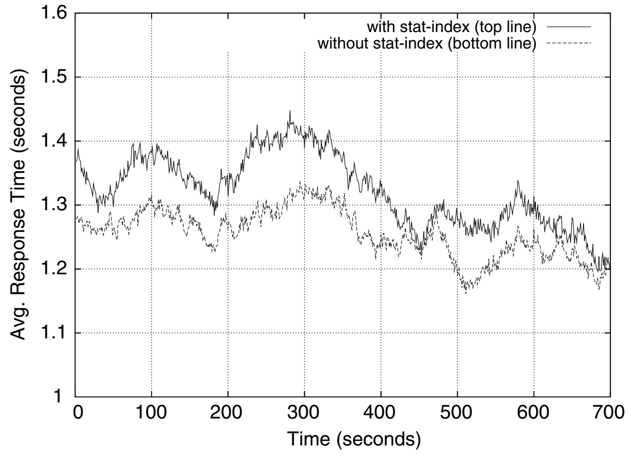


Fig. 17 Throughout the 700 s of steady-state observance, the *stat*-index overhead is approximately 8% under ZIPF($z = 0.8$) query distribution

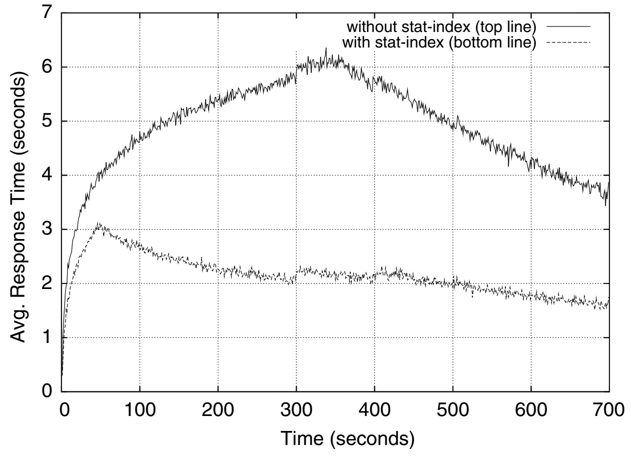


Fig. 18 In a non-balanced system under the ZIPF($z = 0.8$) query distribution the use of the *stat*-index improves ART up to a factor of 3, compensating for the overhead observed in Fig. 17

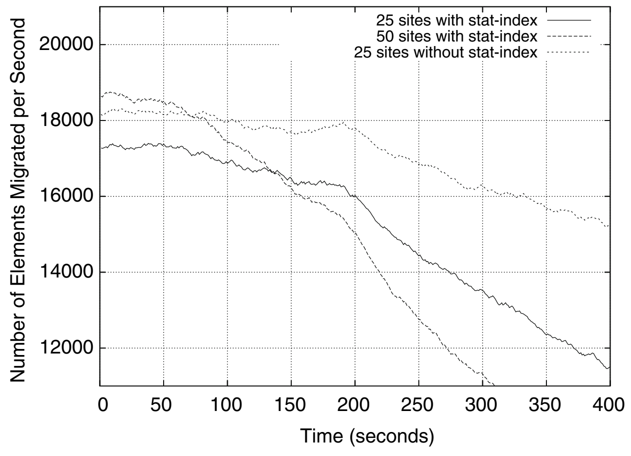


Fig. 19 Amount of data migrated per second

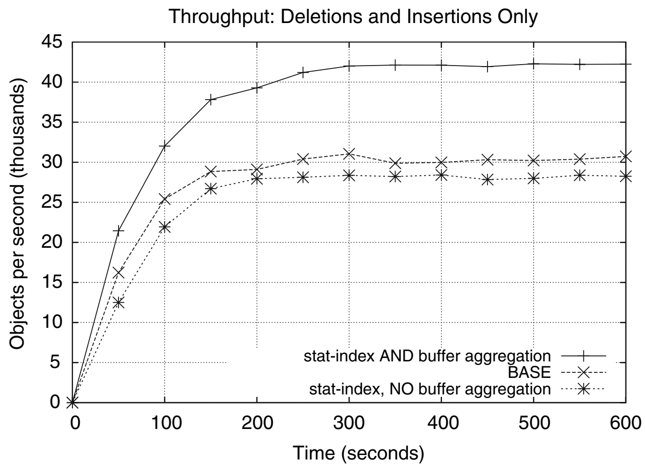
able to quickly reduce the amount of data reorganization by starting with the "hottest" data first. Figure 19 also gives us an insight as to how scalable the system is. We see that with 50 sites, more data is migrated in the beginning when there is a larger pool of underloaded sites to choose from. Thus, a balanced state is reached much sooner. Based on a linear fit of the data, the rate with 50 sites is 2.2 times greater than with 25 sites and 13 times grater than 25 sites with no *stat*-index.

6.4 Throughput analysis

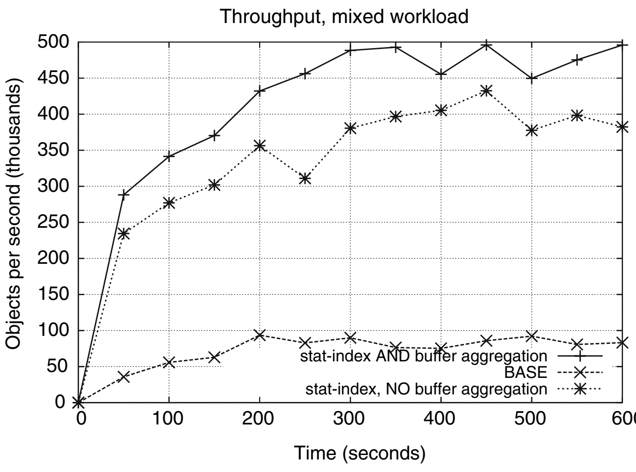
We measure throughput as the number of objects per second delivered by the system in response to queries or deletions/insertions. Any time we issue a deletion and/or insertion, the operation affects a single object. We run experiments with up to 50 machines, which represents our physical limitations. We analyze how such a system performs under two scenarios. In the first case, Fig. 20a, we administer only deletions/insertions to the system. This results in little or no need for load balancing which reveals the overhead of the *stat*-index and the benefit of our query aggregation methods. Compared to the BASE system (where the *stat*-index and query aggregation are turned off), Fig. 20a shows that a system with the *stat*-index but without query aggregation exhibits 8.14% lower throughput. On the other hand, our full-featured system shows an improvement of a factor of 1.36 over the base configuration. The second scenario is more realistic as we mix deletions/insertions and spatial queries with the workload characteristics described in Sect. 6.3, namely ZIPF(z=0.8). As data is fetched from the system, we observe much higher throughput rates than the previous case. This is shown by the graph in Fig. 20b. However, this scenario also shows a significant improvement of the *stat*-index-only system over the base case. This improvement in throughput, a factor of 4.65, is attributed to the fact that the query workloads generate hot-spots that are efficiently balanced out through the use of the *stat*-index. The base system cannot cope with these hot-spots and becomes bottle-necked by the overloaded sites. Finally, the addition of our query aggregation methods improves the throughput of the *stat*-index-only system by 23%. The experiments we have described so far show that the small overhead of maintaining a *stat*-index is compensated for by the performance improvements achieved from utilizing the information provided by the *stat*-index.

6.5 Evaluation of query aggregation methods

Similarly to the *stat*-index, our proposed query aggregation methods exhibit certain CPU overheads at the trade-off of reduced I/O operations. Through empirical observation we show that this overhead is minimal compared to the provided benefits. We study the performance of the proposed query aggregation methods under queries of two spatial distributions: uniform (UNI) and skewed (SKEW), where the skew is Zipfian with zipf factor 0.8. We shall indicate the K-means clustering method as K-DISK, the combined K-means with *stat*-index lookups as K-SI, and the R*-tree method as R*-CLUST. The baseline system processes queries accumulated during the migration process one-at-a-time. As explained in Sect. 5, the AGGR method traverses the R*-tree in a depth-first path, retrieving each R*-tree node that intersects any of the buffered queries. We aggregate the number of I/Os performed (Fig. 21) and the net CPU time (Fig. 22) spent during the processing of query queues through a single experiment and repeat the experiment with skewed data distributions. It should be noted that the CPU time for the AGGR



(a) Updates only



(b) Mixed deletions/insertions and queries

Fig. 20 Throughput of the system where **a** only deletions/insertions are applied to the data set, with buffered inserts bulk-loaded into the R*-tree and **b** deletions/insertions are mixed in with queries arriving every 10 ms with dimensions chosen from a normal distribution with mean side 0.01 units and standard deviation 0.001 units and Zipfian spatial distribution with $z = 0.8$

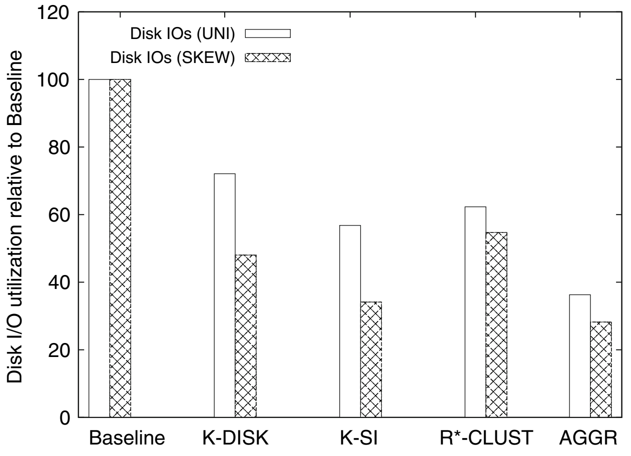


Fig. 21 Disk I/O performance of query aggregation schemes compared to a baseline scenario under uniform (UNI) and skewed (SKEW) data distribution

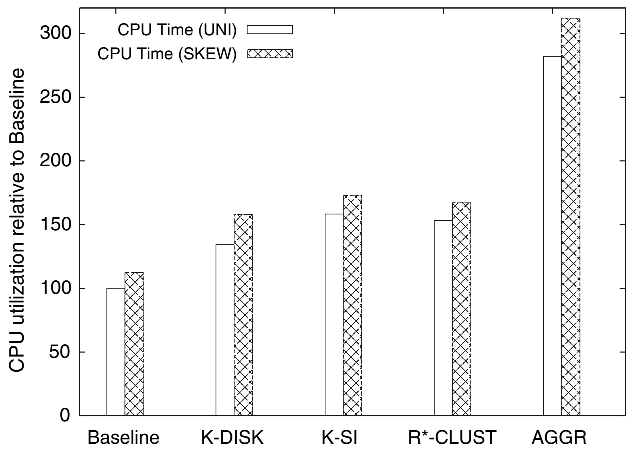


Fig. 22 CPU performance of query aggregation schemes compared to a baseline scenario under uniform (UNI) and skewed (SKEW) data distribution

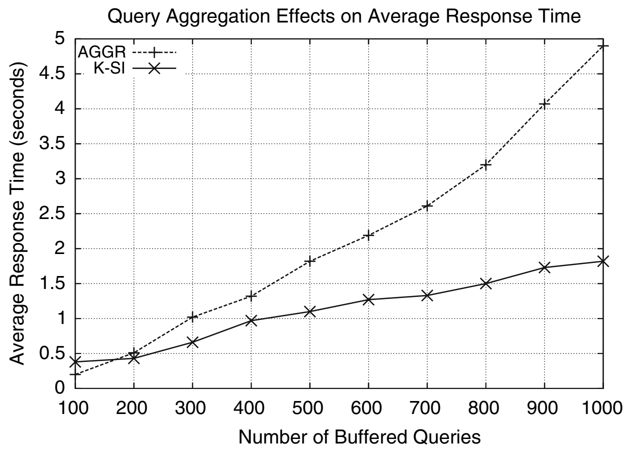


Fig. 23 Average response times for the AGGR and K-SI aggregation algorithms

method is nearly double that of any of the other approaches while its I/O utilization is about 6% lower than K-SI. We carry out the experiment with K-DISK, K-SI and R*-CLUST clustering of accumulated queries in our UltraSPARC COW configuration. Figure 21 shows the performance of our clustering methods relative to AGGR and the baseline scheme. Overall, any of the clustering methods can reduce the number of I/Os for aggregated queries by 28–44% under the UNI query distribution and by 46–66% under the SKEW query distribution. K-SI performs the best overall, and provides the largest gain in difference between the UNI and SKEW distributions. This is the case because skewed data lends itself to high potential for exploit of cache locality in LFRU caching, and the *stat*-index aids in characterizing precisely the access distributions of the data. This comes at the cost of additional CPU cycles, where K-DISK does the best. R*-CLUST on the other hand, spends almost as much CPU time as K-SI, while generating 8% more I/Os. Furthermore, R*-CLUST does not perform well under skewed distributions due to the incurred cost of reinsertions when the data falls in the same leaves. This shows that a K-means clustering scheme in tandem with our *stat*-index can provide significant gains in diminishing the adverse effects of the migration overhead.

Although AGGR produces the lowest disk utilization, its equal treatment of all queries can have a negative effect on the query response time. Namely, all queries will complete at the same time—when the last query has completed. In Fig. 23 we compare the ART (ART) produced when there are 100–1,000 buffered queries for AGGR and K-SI. For small number of queries in the buffer, it is advisable to use AGGR due to its low disk utilization. However, in our infrastructure, when there are more than 180 queries in the buffer, K-SI improves query response time.

6.6 Load distribution of very large data sets

The experiments performed so far assume a small object size of 40 bytes. To evaluate the scalability of our proposed storage manager for data sets of larger object sizes, we vary the object size from 1 to 10 KBytes on clusters of 20 and 40 HP DL360 servers with 2 GBytes of main memory and 2GHz Intel processors.[5] All servers are interconnected with a 1 Gbps switch with 32 Gbps of forwarding capacity. In this set of experiments, we hold the number of objects fixed at 100 million, effectively creating a data set that varies in size from 100 GBytes to 1 TByte. The results for the 20 and 40 HP DL360 servers configurations are depicted in Figs. 24 and 25, respectively. The experiments were initialized with a skewed data distribution such that the standard deviation of the loads amongst the sites was 25%. We measured the time

[5] The 40 HP servers are courtesy of Accepted Ltd, Maroussi, Greece.

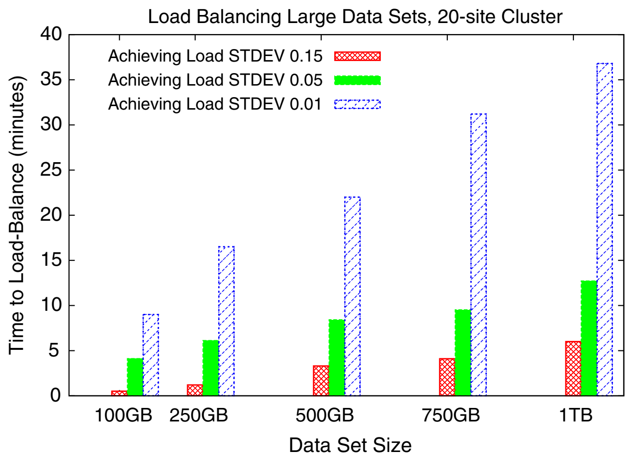


Fig. 24 Effect of object size on the time to perform load-balancing on a cluster of 20 HP servers

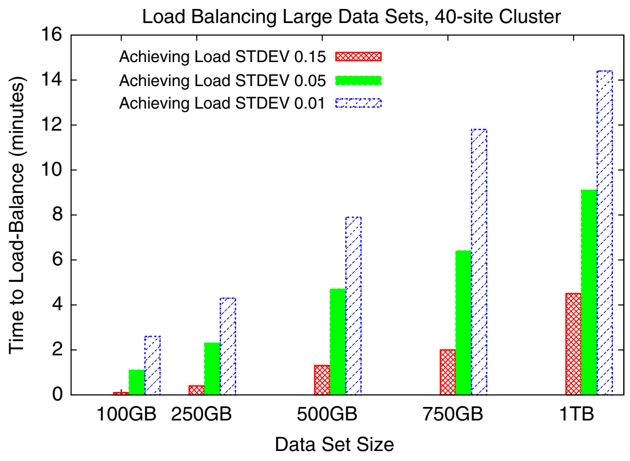


Fig. 25 Effect of object size on the time to perform load-balancing on a cluster of 40 HP servers

it took the system to reach standard deviations of 15, 5 and 1%, under the same workloads detailed in the beginning of this section. Notably, the results show that the load-balancing time increases at a lower rate than the increase of the data set size and object size. For example, a 10-fold increase in object size results in a worst-case load-balancing time increase of a factor of 5.5. This is the case because data redistribution may occur simultaneously between multiple sites in the system. Interestingly, the results also show that the most significant effect of the load balancing is achieved in the first stages of the experiments—the first 10% STDEV decrease takes much less time than for the following 10%. This difference is particularly distinguished in the smaller cluster of 20 sites.

For extremely large sites, with petabytes of data and (tens of) thousands of workstations, we expect our storage manager to scale up very well, especially under the provision of a strong network infrastructure. A good switch would allow multiple gigabit per second connections for simultaneous data transfer between multiple sites [53]. Furthermore, the results in Figs. 13, 14, 24 and 25 indicate that our storage manager can perform faster load balancing when there are more sites available in the cluster.

Lastly, in an environment of varying object sizes, we note that the *stat*-index can be adjusted to include the total object-size in a given subtree. This information could be leveraged by the data-selection process in order to give preference to objects/subtrees of smaller size when performing migration. This extension to the data-selection algorithm is, however, left for future work, as it requires differentiated algorithmic treatment when handling deletions in the *stat*-index.

7 Conclusions and future work

We presented a scalable COW-based storage management system for update-intensive multidimensional data that performs well even at deletion/insertion rates that may make general index structures deficient. At the same time, the system's self-tuning abilities make it devoid of expensive administrative aid and allow it to be self-sustained even under unpredictable changes of access and update patterns. This is accomplished through careful but cost-efficient data redistribution while maintaining short response times. These features are a direct result of a number of novel techniques that provide significant performance improvements over previously proposed architectures:

- dynamic data reorganization achieves load balancing through identification of "hot" spots in the COW and applies cost-driven data selection algorithms to migrate portion of the "hot" data to less loaded sites in the cluster,
- this identification process is accomplished through the use of a highly tunable, *stat*-index of condensed access statistics providing for dynamically adaptable levels of granularity,
- query aggregation mechanisms allow for batch handling of buffered queries and efficient locking operations ensure the consistency of the indexed data,
- distributed collaboration in the self-tuning decision process relieves the central site's responsibility for load balancing initiation, while reducing the system's dependence on hardware parameters of individual sites.

We incorporate all of the above features into a fully fledged prototype used to substantiate our analysis and to provide concrete experimental results on the performance improvements achieved in a highly distributed setting with a data set size reaching 1 TByte. These results show that our COW-based system can easily grow on demand and, thus, is suitable for applications involving very large volumes of frequently modified data such as those encountered in environments with spatio-temporal requirements. Based on the analysis and experimental results, our LAN-based storage manager demonstrates robust performance benefits in highly dynamic settings.

Our future work aims towards an entirely server-less architecture in a hierarchy of clusters distributed over wide-area networks. To deal with the network delays, the storage manager will incorporate efficient synchronization algorithms for data replication. In addition, we plan to provide support for nearest-neighbor, top-*k*, and spatial join queries with the fundamental premise that high update rates would render conventional versions of these distributed algorithms inefficient. Our future work will also focus on provisions for massive streams of spatio-temporal data sets. Furthermore, it may be of interest to explore the possibility of migrating not the "hottest" data, but a set of data which is less "hot," but is still responsible for a significant amount of the load at a site. A preliminary version of this work has appeared in EDBT'04 [27].

Acknowledgments We would like to thank the anonymous reviewers for their valuable comments which helped us improve our work. This research has been sponsored by an US Department of Education GAANN Graduate Fellowship and the Univ. of Athens Research Foundation. We are also thankful to Accepted Ltd in Maroussi, Greece for providing us with use of their hardware.

Appendix A

Space requirements for the *stat*-index

In the context of the R*-tree, we define "elements," as the $\langle mbr, ptr \rangle$ tuples found in each R*-tree node. Therefore, for a fixed fanout f, each node (either data or internal) has f elements. The number of elements at depth d is equal to the number of nodes at depth d times the fanout at that depth. However, the number of nodes at depth d is exactly the number of elements one level higher:

$$\text{Elements}(d) = \text{Elements}(d-1) \times \phi(d)$$

and for $d = 0$ we have $\text{Elements}(0) = f$. Thus, for $\phi(d) = f \times a^d$ (where $a < 1$) we have

$$\text{Elements}(d) = f \times \prod_{k=1}^{d} \phi(k)$$

which gives us

$$\text{Elements}(d) = f \times \prod_{k=1}^{d} fa^k$$

however, using the identity

$$lg\left(\prod_{k=1}^{d} fa^k\right) = \sum_{k=1}^{d} lg(fa^k)$$

$$lg\left(\prod_{k=1}^{d} fa^k\right) = d \times lg(f) + lg(a)\sum_{k=1}^{d} k$$

$$lg\left(\prod_{k=1}^{d} fa^k\right) = lg(f^d) + lg(a)\frac{1}{2}d(d+1)$$

$$lg\left(\prod_{k=1}^{d} fa^k\right) = lg(f^d a^{\frac{1}{2}d(d+1)})$$

$$\prod_{k=1}^{d} fa^k = f^d a^{\frac{1}{2}d(d+1)}$$

and, finally

$$\text{Elements}(d) = f\prod_{k=1}^{d} fa^k = f^{d+1}a^{\frac{1}{2}d(d+1)}$$

Thus, the total number of elements in a tree of height H is the summation over the number of elements at each depth:

$$\text{TotalElements}(H) = \sum_{k=0}^{H} f^{k+1}a^{\frac{1}{2}k(k+1)}$$

References

1. Arge, L., Hinrichs, K., Vahrenhold, J., Vitter, J.: Efficient bulk operations on dynamic R-trees. In: ALENEX, pp. 328–348 (1999)
2. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. Atlantic City, 23–25 May, pp. 322–331. ACM Press, New York (1990)
3. Berchtold, S., Böhm, C., Kriegel, H.: Improving the query performance of high-dimensional index structures by bulk-load operations. In: Proceedings of the 6th International Conference on Extending Database Technology, EDBT. vol. 1377, pp. 216–230. Springer, Heidelberg, 23–27 (1998)
4. Bruno, N., Chaudhuri, S., Gravano, L.: STHoles: a multidimensional workload-aware histogram. In: SIGMOD Conference, pp. 211–222 (2001)
5. Chen, L., Choubey, R., Rundensteiner, E.: Bulk-insertions infor-trees using the small-tree-large-tree approach. In: ACM-GIS '98, Proceedings of the 6th International Symposium on Advances in Geographic Information Systems, 6-7 November 1998, Washington, pp. 161–162. ACM, New York (1998)
6. Choubey, R., Chen, L., Rundensteiner, E.: GBI: a generalized R-tree bulk-insertion strategy. In: Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, July 20-23 July 1999, Proceedings, vol. 1651. Lecture Notes in Computer Science, pp. 91–108. Springer, Heidelberg (1999)
7. Deshpande, P., Ramasamy, K., Shukla, A., Naughton, J.F.: Caching multidimensional queries using chunks. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, 2–4 June, pp. 259–270 (1998)

8. Eager, D.L., Lazowska, E.D., Zahorjan, J.: A comparison of receiver-initiated and sender-initiated adaptive load sharing. In: Proceedings of ACM SIGMETRICS, pp. 1–3 (1985)
9. Ellis, C.: Distributed data structures: a case study. IEEE Trans. Comput. **34**(12), 1178–1185 (1985)
10. Feeley, M., Morgan, W.E., Pighin, F.H., Karlin, A.R., Levy, H.M.: Implementing global memory management in a workstation custer. In: Proceedings of the 21st Symposium on Operating Systems Principles, October (1995)
11. Ghanem, T.M., Shah, R., Mokbel, M.F., Aref, W.G., Vitter, J.S.: Bulk operations for space-partitioning trees. In: ICDE, pp. 29–41 (2004)
12. Gibbons, P.B., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. ACM Trans. Database Syst. **27**(3), 261–298 (2002)
13. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan-Kaufman, San Mateo (1992)
14. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD'84, Proceedings of Annual Meeting, Boston, 18-21 June, pp. 47–57. ACM Press, New York (1984)
15. Hadjielefteriou, M.: R*-tree implentation version 0.62b. http://ucr.ca.edu/marios
16. Hadjieleftheriou, M., Kriakov, V., Tao, Y., Kollios, G., Delis, A., Tsotras, V.J.: Spatio-temporal data services in a shared-nothing environment. In: Proceedings of 16th International Conference on Scientific and Statistical Database Management SSDBM. June (2004)
17. Hall, J., Hartline, J., Karlin, A.R., Saia, J., Wilkes, J.: On algorithms for efficient data migration. In: 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)
18. Iyengar, S., Bastani, F., Yen, I.: Concurrent maintenance of data structures in a distributed environment. Comput. J. **31**(12), 165–174 (1988)
19. Johnson, T., Krishna, P., Colbrook, A.: Distributed indices for accessing distributed data. In: IEEE Symposium on Mass Storage Systems (MSS '93), pp. 199–208, Los Alamitos, Ca., USA, April. IEEE Computer Society Press (1993)
20. K-means clustering algorithm. http://mathworld.wolfram.com/K-MeansClusteringAlgorithm.html
21. Kamel, I., Faloutsos, C.: Parallel R-trees. In: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, 2-5 June, pp. 195–204. ACM Press, New York (1992)
22. Kamel, I., Faloutsos, C.: Hilbert R-tree: an improved R-tree using fractals. In: VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, 12-15 September 1994, Santiago de Chile, Chile, pp. 500–509. Morgan Kaufmann, Los Altos (1994)
23. Karlsson, J.S.: hQT*: a scalable distributed data structure for high-performance spatial accesses. In: FODO, pp. 37–46 (1998)
24. Khuller, S., Kim, Y.-A., Wan, Y.-C.J.: Algorithms for data migration with cloning. In: Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. pp. 27–36. ACM Press, New York
25. Kornacker, M., Banks, D.: High-concurrency locking in R-trees. In: VLDB'95, Zurich, pp. 134–145 (1995)
26. Koudas, N., Faloutsos, C., Kamel, I.: Declustering spatial databases on a multi-computer architecture. In: Advances in Database Technology—EDBT'96, 5th International Conference on Extending Database Technology, Avignon (1996)
27. Kriakov, V., Delis, A., Kollios, G.: Management of highly dynamic multidimensional data in a cluster of workstations. In: Proceedings of the 9th International Conference on Extending Database Technology, EDBT, vol. 2992, pp. 748–764. Springer, Heidelberg (2004)
28. Kroll, B., Widmayer, P.: Distributing a search structure among a growing number of processors. In: Proceedings of the 1994 ACM SIGMOD Conference, pp. 265–276 (1994)
29. Kulkarni, P., Ganesan, D., Shenoy, P.J., Lu, Q.: *SensEye*: a Multi-Tier Camera Sensor Network. In: ACM Multimedia, pp. 229–238, (2005)
30. Lee, M., Kitsuregawa, M., Ooi, B., Tan, K., Mondal, A.: Towards self-tuning data placement in parallel database systems. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 16–18 May 2000, Dallas, pp. 225–236 (2000)
31. Litwin, W., Neimat, M.A.: k-RP*S: a scalable distributed data structure for high-performance multi-attribute access. In: Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, 18-20 December, Miami Beach, pp. 120–131. IEEE Computer Society (1996)
32. Litwin, W., Neimat, M.A., Schneider, D.: Linear hashing for distributed files. In: Proceedings of the 1993 SIGMOD Conference, Washington, May (1993)
33. Matsliach, G., Shmueli, O.: An efficient method for distributing search structures. In: First International Conference on Parallel and Distributed Information Systems, pp. 159–166 (1991)
34. Mondal, A., Kitsuregawa, M., Ooi, B.C., Tan, K.L.: R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In: Proceedings of ACM Geographic Information Systems, pp. 28–33. ACM Press, New York (2001)
35. Ousterhout, J. K. G. T. Hamachi, Mayo, R.N., Scott, W.S., Taylor, G.S.: Magic: A VLSI layout system. In: 21st Design Automation Conference, pp. 152–159, June (1984)
36. Pagel, B., Korn, F., Faloutsos, C.: Deflating the dimensionality curse using multiple fractal dimensions. In: ICDE, pp. 589–598 (2000)
37. Panagos, E., Biliris, A.: Synchronization and recovery in a client-server storage system. VLDB J. **6**(3), 209–223 (1997)
38. Panwar, S., Mao, S., Ryoo, J., Li, Y.: TCP/IP Essentials: A Lab-Based Approach. Cambridge University Press, Cambridge (2004)
39. Papadopoulos, A., Manolopoulos, Y.: Nearest neighbor queries in shared-nothing environments. GeoInformatica **1**(4), 369–392 (1997)
40. Papadopoulos, A., Manolopoulos, Y.: Parallel bulk-loading of spatial data. Parallel Comput. **29**(10), 1419–1444 (2003)
41. Patel, J., Yu, J.-B., Kabra, N., Tufte, K.: Building a scalable geospatial dbms: technology, implementation, and evaluation. In: Proceedings of the ACM SIGMOD, pp. 336–347 (1997)
42. Porkaew, K., Lazaridis, I., Mehrotra, S.: Querying mobile objects in spatio-temporal databases. In: Proceedings of 7th SSTD, July (2001)
43. Pritchett, D.: BASE: An ACID Alternative. ACM Queue, 6(3), May/June (2008)
44. Qiao, L., Iyer, B.R., Agrawal, D., El Abbadi, A.: Automated storage management with QoS guarantee in large-scale virtualized storage systems. IEEE Data Eng. Bull. **29**(3), 47–54 (2006)
45. Ramamritham, K., Chrysanthis, P.K.: A taxonomy of correctness criteria in database applications. VLDB J. **5**(1), 85–97 (1996)
46. Robinson, J.T.: The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. In: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, 29 April–1 May, pp. 10–18. ACM Press, New York (1981)
47. Roussopoulos, N., Kotidis, Y., Roussopoulos, M.: Cubetree: Organization of and bulk updates on the data cube. In: SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, 13-15 May 1997, Tucson, pp. 89–99. ACM Press, New York (1997)
48. Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M.A.: Indexing the positions of continuously moving objects. In: Proceeding of the ACM SIGMOD, pp. 331–342, May (2000)

49. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Comput. Surv. **31**(2), 158–221 (1999)
50. Scheuermann, P., Weikum, G., Zabback, P.: Data partitioning and load balancing in parallel disk systems. VLDB J. 7(1) (1998)
51. Schnitzer, B., Leutenegger, S.: Master-Client R-Trees: A new parallel R-tree architecture. In: Statistical and Scientific Database Management, pp. 68–77 (1999)
52. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R+-tree: a dynamic index for multi-dimensional objects. In: VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, 1–4 September 1987, Brighton, pp. 507–518. Morgan Kaufmann, Los Altos (1987)
53. Smiljanic, A.: Flexible bandwidth allocation in high-capacity packet switches. IEEE/ACM Trans. Netw. **10**(2), 287–293 (2002)
54. Sun, X., Wang, R., Salzberg, B., Zou, C.: Online B-tree merging. In: Proceedings of ACM SIGMOD, pp. 335–346 (2005)
55. Szalay, A., Gray, J., van den Berg, J.: Petabyte scale data mining: dream or reality. In: Proceedings of SIPE Astronomy Telescopes and Instruments, August (2002)
56. Slutz, D., Barclay, T., Gray, J.: TerraServer: a spatial data warehouse. In: Proceedings of ACM SIGMOD, pp. 307–318 (2000)
57. Satoh T., Honishi, T., Inoue, U.: An index structure for parallel database processing. In: IEEE Second International Workshop on Research Issues on Data Engineering, pp. 224–225 (1992)
58. The Earth Observing System Data and Information System. http://spsosun.gsfc.nasa.gov/eosinfo/EOSDIS_Site/index.html
59. Tao, Y., Papadias, D.: Range aggregate processing in spatial databases. IEEE Trans. Knowl. Data Eng. **16**(12), 1555–1570 (2004)
60. Thaper, N., Guha, S., Indyk, P., Koudas, N.: Dynamic multidimensional histograms. In: SIGMOD Conference, pp. 428–439 (2002)
61. Theodoridis Y., Sellis T. (1996) A model for the prediction of R-tree performance. In: PODS, pp. 161–171 (1996)
62. Van den Bercken, J., Seeger, B.: An evaluation of generic bulk loading techniques. In: VLDB'01, Roma, pp. 461–470 (2001)
63. Van den Bercken, J., Seeger, B., Widmayer, P.: A generic approach to bulk loading multidimensional index structures. In: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, 25–29 August, Athens, pp. 406–415. Morgan Kaufmann, Los Altos (1997)
64. Zeiler, T.L.: LANDSAT program report 2002. Technical report, US Geological Survey—US Department of Interior. EROS Data Center, Sioux Falls (2002)
65. Zou, C., Salzberg, B.: On-line reorganization of sparsely-populated B+trees. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, 4–6 June, pp. 115–124. ACM Press, New York (1996)
66. Zou, C., Salzberg, B.: Safely and efficiently updating references during on-line reorganization. In: Proceedings of VLDB, pp. 512–522 (1998)