Workload-Based Generation of Administrator Hints for Optimizing Database Storage Utilization

KAUSHIK DUTTA, RAJU RANGASWAMI, and SAJIB KUNDU
Florida International University

16

Database storage management at data centers is a manual, time-consuming, and error-prone task. Such management involves regular movement of database objects across storage nodes in an attempt to balance the I/O bandwidth utilization across disk drives. Achieving such balance is critical for avoiding I/O bottlenecks and thereby maximizing the utilization of the storage system. However, manual management of the aforesaid task, apart from increasing administrative costs, encumbers the greater risks of untimely and erroneous operations. We address the preceding concerns with STORM, an automated approach that combines low-overhead information gathering of database access and storage usage patterns with efficient analysis to generate accurate and timely hints for the administrator regarding data movement operations. STORM's primary objective is minimizing the volume of data movement required (to minimize potential down-time or reduction in performance) during the reconfiguration operation, with the secondary constraints of space and balanced I/O-bandwidth-utilization across the storage devices. We analyze and evaluate STORM theoretically, using a simulation framework, as well as experimentally. We show that the dynamic data layout reconfiguration problem is NP-hard and we present a heuristic that provides an approximate solution in $O(Nlog(\frac{N}{M}) + (\frac{N}{M})^2)$ time, where M is the number of storage devices and N is the total number of database objects residing in the storage devices. A simulation study shows that the heuristic converges to an acceptable solution that is successful in balancing storage utilization with an accuracy that lies within 7% of the ideal solution. Finally, an experimental study demonstrates that the STORM approach can improve the overall performance of the TPC-C benchmark by as much as 22%, by reconfiguring an initial random, but evenly distributed, placement of database objects.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management

This work is sponsored in part by NSF grant IIS-0534530 and DoE grant ER25739.
This is an extended version of a paper titled "STORM: An Approach to Database Storage Management in Data Center Environments," by K. Dutta and R. Rangaswami, which appeared in *Proceedings of the IEEE International Conference on Cluster Computing and the Grid*, May 2007. ©2007 IEEE.
Authors' addresses: K. Dutta (contact author), College of Business, Florida International University, 11200 S.W. 8th St., Miami, FL 33199; email: Kaushik.Dutta@fiu.edu; R. Rangaswami, S. Kundu, School of Computing, Florida International University, 11200 S.W. 8th St., Miami, FL 33199; email: {raju,skund001}@cs.fiu.edu.

General Terms: Design, Algorithms, Performance

ACM Reference Format:

Dutta, K., Rangaswami, R., and Kundu, S. 2008. Workload-based generation of administrator hints for optimizing database storage utilization. ACM Trans. Stor. 3, 4, Article 16 (February 2008), 28 pages. DOI = 10.1145/1326542.1326545 http://doi.acm.org/10.1145/1326542.1326545

1. INTRODUCTION

Data center services for medium-to-large enterprises typically host several petabytes of data on disk drives. Most of this storage houses data residing in tens to hundreds of databases. This data landscape is both growing as well as dynamic; new data-centric applications are constantly added at data centers, while regulatory requirements such as SOX [Karl Nagel Corporation 2006] prevent old and unused data from being deleted. Further, the data access characteristics of these applications change constantly. Ensuring peak application throughput at data centers is incumbent upon addressing this dynamic data management problem in a comprehensive fashion.

IT managers have various storage options, ranging from low-cost SATA, and multidisk RAIDs to high-end custom storage solutions. To accommodate the rapid growth in the number of storage devices across data centers and their associated management overhead,[1] data center managers are more and more inclined toward isolating storage management at data centers using storage area networks (SANs [IBM 2006]): a network whose primary purpose is the transfer of data between computer systems and storage elements. Application or database servers connect to storage devices through SAN switches or routers [McDATA Corporation 2006].

Although SANs allow significant isolation of storage management from server management, the storage management problem is still complex. Due to the dynamic nature of modern enterprises, the interaction and use of applications, and even the data associated with a single application, changes over time. Dynamic changes in the set of "popular" data results in a skewed utilization of network storage devices, both in terms of storage space and I/O bandwidth. In statically allocated storage systems, such skewed storage utilization eventually degrades the performance of applications, creating the necessity to buy more storage (when existing storage is not fully utilized), thereby resulting in overall cost increment.

The disadvantages of an static storage allocation have been long recognized by data center administrators. They regularly spend copious amounts of time moving data between storage devices to avoid such skewness. However, optimal data movement is a complex problem that entails obtaining accurate knowledge of data popularity at the right granularity and choosing from an exponential number of possible target solutions, while ensuring that the volume of data

[1]Recent estimates put expenditure on storage management at approximately one person per 1–10 TB and these estimates a state that storage cost is dominated by storage management cost rather than hardware cost over the long term [Allen 2001; Lamb 2001].

moved is minimal. As a result, manual decision making in large data centers containing several terabytes of data and hundreds of storage devices (if not thousands) is time consuming, inefficient, and at best results in suboptimal decisions. Off-the-shelf relational databases contribute to a large portion of these terabytes of data, and the manual data management tasks of system administrators mostly involve the remapping of database elements (tables, indexes, logs, etc.) to storage devices.

In this article, we present the architecture and design of Storm, a system that performs timely and automatic identification of skewness in database storage utilization in a data center environment and that accordingly proposes a near that optimal data movement strategy. Moving a large amount of data between storage devices requires considerable storage bandwidth and time, and although such movement is typically done in periods of low activity such as night-time, it nevertheless runs the risk of affecting the performance of applications. Moreover, such data movement operations are so critical that they are seldom done in unsupervised mode; a longer time implies greater administrator cost. A longer time requirement for the data movement also prompts data center managers to postpone such activities and to live with skewed usage for as long as possible. It is therefore critical to minimize the overall data movement in any reconfiguration operation. Storm addresses the problem of reconfiguration with the primary objective of minimizing total data movement, with the secondary objective of balancing the I/O-bandwidth utilization of the storage devices in a SAN system, given storage-device-capacity constraints.

The specific contributions of this article are as follows.

1. We present the architecture, design, and implementation of Storm, an automated tool that aids system administrators in the database management task of optimizing storage utilization.
2. We mathematically model the aforementioned problem with the objective of minimizing the total data movement, given storage-device constraints, and show that it is NP-hard;
3. We propose a two stage greedy heuristic algorithm that provides an acceptable approximate solution. The heuristic tries to move objects of smaller size before choosing to move larger objects (i.e., greedy on size) from storage nodes with higher bandwidth utilizations to storage nodes with lower bandwidth utilization (i.e., greedy on I/O bandwidth utilization).
4. We conduct a simulation study to demonstrate the efficiency and accuracy of the heuristic algorithm across a wide range of database and storage configurations.
5. We conduct an experimental study using the TPC-C benchmark to evaluate the impact of Storm on the overall performance of the database system.

The rest of the article is organized as follows. We examine related research in Section 2. We then present a practical system architecture that incorporates Storm within typical NAS-based database storage environments in Section 3. In Section 4, we theoretically model the problem of dynamic database storage management and present a heuristic solution to this problem in Section 5.

Section 6 presents techniques used in Storm for monitoring database and storage usage patterns. Section 7 presents an evaluation of Storm using a simulation study, comparing it against a baseline optimal solution. In Section 8, using the TPC-C benchmark, we evaluate an implementation of Storm in its ability to improve both the I/O-bandwidth utilization of the storage system and the overall performance of the database system. We make final remarks in Section 9.

2. RELATED WORK

In the research literature, there has been substantial work on automated storage management—too many to list comprehensively here—several of which do suggest databases as a target application. To follow, we list work in the areas of database-aware storage, parallel and distributed databases, and application-independent techniques for storage management, including online data reconfiguration and load balancing.

Sivathanu et al. [2005] propose building database-aware semantically-smart storage systems to improve fault tolerance, availability, and reliability. To accomplish this goal, the authors suggest minimally modifying database implementations to export access statistics. Our work shares the general philosophy of building database-aware storage systems, with the difference that we address a different problem of automatic data movement for improving storage utilization. Further, our techniques derive richer intelligence than the aforementioned work, using explicit querying and passive monitoring that do not require any modifications to existing database systems.

Research on data placement in parallel database systems [Hua and Lee 1990; Mehta and DeWitt 1997; Furtado 2004] may seem related at first glance. However, data placement in a parallel database system is designed with the motivation of achieving maximum query parallelism for a single database system. The goal in our work is to balance the utilization of shared storage devices in a SAN across multiple database systems as typical in a data center setting. We accomplish this by automatically identifying skewness in storage utilization as well as database object popularity, and by utilizing this information to suggest optimal data movement.

Distributed data storage systems such as Mariposa [Stonebraker et al. 1994] have developed ways to place data that is distributed in geographical locations based on access patterns and other cost factors, such as network cost. The basic objectives of Mariposa and our system are different in that Mariposa works on a single distributed database system, while our system works on multiple centralized database systems that store data in a shared SAN environment. Further, Mariposa optimizes for a WAN setting, where network bandwidth is scarce and data are allocated based on optimal network usage. In our system, database servers are connected to SAN devices over a high-speed network. In such a scenario we can assume all storage-devices are equally accessible by database servers. We address the problem of balancing the storage-device utilization which is more applicable in a data center environment.

Several researchers have addressed orthogonal problems related to online reconfiguration of data on disk drives. Most of the proposed solutions in this

space can be used in conjunction with Storm, or during the reconfiguration operation following Storm hints. In Petal [Lee and Thekkath 1996], the idea of logically separating virtual disk addresses to physical locations was proposed to address transparency in scaling capacity and performance as well as for dynamic load-balancing. Zhang et al. [2007] also propose scaling round-robin striped volumes efficiently with reduced impact on foreground workload. Wu and Burns [2005] address the problem of load-balancing data access on storage servers using hashing-based randomized mapping of the source domain to dynamically reconfigurable target domains that map directly to storage servers. Khuller et al. [2003] provide load balance across storage devices by migrating data at the block level. In comparison to these efforts, Storm proposes data movement at the application granularity of data objects, whereby future utilization can be estimated with greater accuracy and storage administrators can enjoy a greater control on the location of such database objects.

Chenyang et al. address reconfiguration-related issues and advocate a control-theoretic approach for achieving a statistical bound on the impact of reconfiguration on foreground I/Os. Lu et al. [2002]; Feng and Zhang [2005] address consistency issues during reconfiguration. Qiao et al. [2006] address the problem of delivering QoS for foreground I/Os during reconfiguration by proposing an optimal schedule for data movement between disk drives. Storm implicitly reduces the impact on foreground workload by minimizing the total data movement during reconfiguration as a primary objective.

Among industry efforts, storage management vendors such as Veritas [2005], Computer Associates [2005], and BMC Software [2005] provide application-independent software for storage management. These solutions typically work at the block level and involve moving blocks of data from one storage device to another to achieve balanced utilization. However, such movement is carried out without semantic knowledge of block content or utilization. Further, such movement may lead to a single database table being arbitrarily split across several drives, severely complicating the task of a database administrator. Our research focuses on data movement at its application-level granularity such as database tables or indices, thereby also utilizing the semantic knowledge of the data being moved. With Storm, system administrators have full control on the location of each database object in the storage system.

Oracle's automatic storage manager (ASM) solution [Transaction Processing Performance Council (TPC) 2006] proposes a different approach to database storage management, by striping each file within a single database across all the available storage using a 1MB stripe size. The claim with ASM is that it eliminates the need to move data dynamically because the striped layout of each file across all drives implicitly balances I/O load. However, this solution works on a per-DB level, requiring a dedicated "disk group" to be allocated to each database. Our solution works in an environment where sharing storage across multiple databases is critical, thereby ensuring better utilization of storage resources.

Lastly, load-balancing server resource usage has been an active area of research for over a decade, since early work on web servers [Kwan et al. 1995]. In the storage domain, disk striping and replication [Patterson et al. 1988; Ganger

et al. 1993] to distribute disk accesses across a set of drives has been popular for quite some time. Traditional load balancers work in dynamic fashion, operating at a per-request level. Further, they have a single objective, namely that of balancing the request load of a set of servers/disk drives. We address data movement for balancing data-access load with the dual objectives of minimizing data movement and nullifying skewness in storage utilization, all while meeting the projected capacity requirement based on future growth of data. Further, such data movement is performed periodically with a much coarser time interval, rather than in a continuous fashion.

3. SYSTEM DESIGN AND ARCHITECTURE

We consider a data center environment with a tiered architecture for providing services, comprised of application servers, database servers, and storage nodes. At the head are the application servers, which service user requests. Application servers use the database servers to query the databases, which in turn access data from the tables stored in the SAN device pool. Further, we assume that the data center comprises several database server clusters; these clusters share the storage space provided by the SAN device pool.

Our work focuses on the interaction between database servers and SAN devices. Database servers allocate storage space in the SAN devices to store database objects such as tables, indexes, logs, etc. The access patterns for individual objects managed by the database servers are application dependent and can vary widely over time. Consequently, one of the key problems in managing SAN devices in a database-centric system is that of moving data from one storage device to another to accommodate the future growth of objects and to balance the utilization of individual devices in the SAN. This primarily includes reconfiguring the storage allocation and data placement on a per-object basis, based on application access patterns. Successful reconfiguration leads to more balanced access to the SAN device pool by the database servers, thereby preventing bottlenecks at individual storage nodes.

The storage management tasks required for such reconfiguration include information gathering, analysis, decision making, and execution, conforming to the monitor-analyze-plan-execute loop proposed in the IBM's Autonomic Computing intiative [Kephart and Chess 2003]. Currently, each of these tasks are performed manually by system administrators based on human intuition and experience. Storm performs data gathering, analysis, and decision making automatically, based on which data center managers can choose to reconfigure the layout of objects to improve the storage utilization. It is important to point out that the reconfiguration process could be automated as well, but doing so would require additional mechanisms to ensure data and operational consistency during the reconfiguration operation. Another psychological hurdle is that administrators are typically reluctant to accept automatic and unsupervised data reconfiguration. Hence, Storm merely suggests favorable data movement operations and leaves the decisions of "if" and "when" to the administrator.

The key component of our system is the *TableMapper*, which has access to the database servers as well as to the SAN device pool. Figure 1 depicts

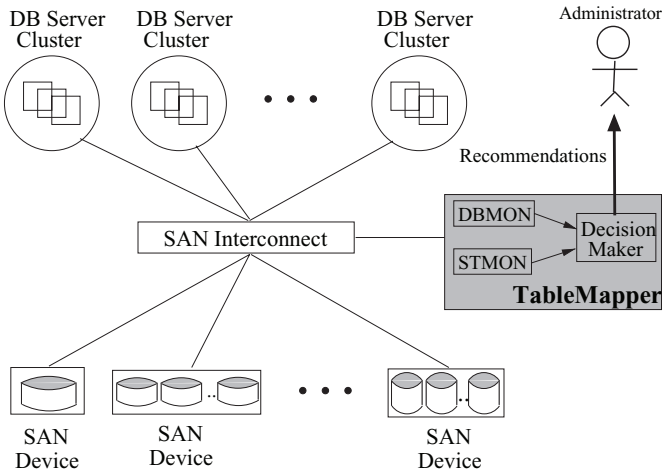

Fig. 1. Storm in a SAN configuration connecting the database servers and SAN devices.

the TableMapper in a SAN configuration that connects the DB server clusters to storage devices in the SAN. In NAS/SAN/object-based storage systems [Mesnier et al. 2003] and even in specialized large-scale systems such as the IBM Storage Tank [Menon et al. 2003], the TableMapper could be colocated with the metadata server so that partial information can be obtained locally with minimum overhead. The TableMapper gathers object access and storage usage data (elaborated in Section 6) using the *database monitor* (DBMON) and *storage monitor* (STMON) modules, analyzes the data, and makes reconfiguration decisions within its *decision maker* module. The STMON component gathers data related to storage devices such as storage capacity and I/O bandwidth. These items of storage data are static in nature and can be adjusted manually when a new storage is added, or existing storage is taken out from SAN. The storage utilization, that is, which database object is using which storage node and how much space it is consuming, is also gathered by the STMON from database systems. The DBMON component gathers usage information of key database objects (tables and indices). The data gathering mechanism of both the DBMON and STMON components are described in detail in Section 6. Based on this data, the decision maker analyzes and makes reconfiguration decisions. This analysis and the decision making process are elaborated in Sections 4 and 5. In case a reconfiguration is deemed appropriate by the decision maker, it notifies the system administrator, who may choose to act upon the recommendation.

In realizing the TableMapper we had to consider two key factors. First, the TableMapper must be nonintrusive in collecting object and storage usage information from the database servers. Since this operation is performed periodically and infrequently, it can be performed during system idle-time. The second challenge is to avoid a bottleneck at the TableMapper itself. We argue that since the TableMapper only manages metadata and the actual dataflow bypasses the TableMapper, it is unlikely to become a bottleneck. Further, following our

architecture, we envision no significant hurdles to using multiple DBMONs to collect data from a number of database servers and storage nodes.

The decision maker module of the TableMapper is its most complex component. Based on gathered database-object and storage usage data, the decision maker proposes a reconfiguration of object layout on storage devices. In doing so, it must balance multiple optimization objectives. First, the new configuration should be achievable with minimal data movement. Reducing the total amount of data movement will contribute to realizing the new configuration in less time and thus at lower cost. This will result in reducing the amount of network traffic, and also reducing the volume of data which may be potentially rendered unavailable during the move. These factors would also encourage data center IT managers to perform more frequent reconfiguration, leading to reduced skewness in storage usage over time. Second, it must ensure that none of the storage devices is overly utilized in terms of I/O bandwidth. This is addressed by posing a reconfiguration constraint so that the percentage of I/O-bandwidth utilization for each device is below the average percentage I/O-bandwidth utilization across all storage devices plus a small configurable threshold. Finally, the new configuration should support future table growth until the storage managers decide for another round of reconfiguration.

In the next section, we formally describe the dynamic storage reconfiguration decision making problem, followed by a heuristic solution (in Section 5) for such decisions that will help data center storage managers.

4. MODEL

We describe the configuration decision-making problem formally as *"Given a set of database objects J with their present growth rate g_j, usage characteristic r_j and size s_j, given a set of network storage I with allowable I/O-bandwidth U_i^m and capacity B_i specifications, and given the present assignment c_{ij} of database objects to storage nodes, determine a new assignment x_{ij} of objects to storage nodes that:* (i) *will result in minimal physical movement of data across storage devices to realize the new assignment;* (ii) *will balance the I/O-bandwidth utilization of storage nodes; and* (iii) *that will meet the future size growth of objects for certain time T."*

Table I describes the parameters of the proposed model. Based on these we formulate the dynamic data layout reconfiguration problem **P**, as follows.

Problem P.

$$\mathbf{Z(P)} = min \sum_i \sum_j (c_{ij} - x_{ij}) c_{ij} s_j \tag{1}$$

subject to

$$\sum_j (s_j + T g_j) x_{ij} \leq B_i \qquad \forall i \tag{2}$$

$$\bar{U} = 100 \frac{\sum_j \sum_i c_{ij} r_j}{\sum_i U_i^m} \tag{3}$$

Table I. Model Parameters

Parameter	Description
i	Index for set of physical storage devices (I)
j	Index for set of database objects, tables and indices (J)
c_{ij}	Equals 1, if j is currently located in storage i 0, otherwise
s_j	Current size of object j in bytes
g_j	Current growth rate of object j in bytes/days
U_i^m	Maximum I/O bandwidth utilization of the storage in bytes/sec
B_i	Storage capacity in bytes for storage i
r_j	The average bytes/sec retrieved from object j j to serve database requests related to object j
U_{th}	Threshold in percentage that can be allowed for a storage to be over-utilized than the average utilization
T	Validity duration of new object location in days
$\bar{U}$	Average percent utilization of all storages
x_{ij}	Equals 1, if the new allocation of object j is to storage i 0, otherwise

$$100\frac{\sum_j x_{ij} r_j}{U_i^m} \leq \bar{U} + U_{th} \qquad \forall i \tag{4}$$

$$\sum_i x_{ij} = \sum_i c_{ij} \qquad \forall j \tag{5}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j$$

$$c_{ij} \in \{0, 1\} \qquad \forall i, j,$$

where **Z(P)** is the optimal value of data movement.

The objective function (1) minimizes the total data movement across storage devices. Constraint (2) ensures that allocated objects have the flexibility to accomodate projected future growth without relocating it to another storage device in the future (for T days). Eq. (3) computes the average percentage of utilization across all storage devices. Constraint (2) ensures that the utilization of each storage node is below the average utilization with a leeway threshold of U_{th}. Constraint (5) ensures that each and every object is assigned to a storage node under the new allocation scheme.

In the current formulation of the problem, we try to balance the I/O bandwidth utilization across storage nodes within the available storage limitation of each node. However, this may lead to unbalanced utilization of storage spaces across various storage nodes. This may easily be addressed by another set of constraints very similar to Constraints 4 for storage sizes. However, to keep the focus of the paper and discussion simple, we do not address this issue in this paper.

THEOREM 1. *The problem **P** is NP-hard.*

PROOF. We show that problem **P** is NP-hard by showing that a special case of the problem reduces to the multidemand constraint, multidimensional

knapsack problem (MDMKP) [Chu and Beasley 1998; Cappanera and Trubian 2005], which is known to be NP-hard.

The MDMKP problem can be stated in math-programming form as

$$max \sum_{j=1,\dots,n} p_j w_j \tag{6}$$

subject to

$$\sum_{j=1,\dots,n} a_{ij} w_j \leq b_i, \forall i = 1, \dots, \forall j = 1, \dots, n \tag{7}$$

$$\sum_{j=1,\dots,n} a_{ij} w_j \geq b_i, \forall i = m+1, \dots, m+q, \forall j = 1, \dots, n \tag{8}$$

$$w_j \in 0, 1, \tag{9}$$

where p_j, a_{ij}, and b_i are some positive real numbers.

Let us assume, $y_{ij} = x_{ij} - c_{ij}$, then the problem **P** can be written as that.

$$max \sum_i \sum_j y_{ij} c_{ij} s_j$$

subject to

$$\sum_j a_j^1 y_{ij} \leq b_i^1 \qquad \forall i,$$

where $a_j^1 = s_j + T g_j$ and $b_i^1 = B_i - \sum_j s_j T g_j$ and

$$\sum_j a_j^2 y_{ij} \leq b_i^2 \qquad \forall i,$$

where $a_j^2 = r_j$ and $b_j^2 = (\bar{U} + U_{th}) U_i^m / 100 + \sum_j c_{ij} r_j$ and

$$y_{ij} \leq 0, \qquad y_{ij} \geq 0 \qquad \forall i, j.$$

The preceding form of the problem **P** clearly maps to the MDMKP. Thus we can say that the problem can be reduced to an MDMKP in polynomial time. So the problem **P** is also NP-hard. □

Note that the previous problem **P** is an integer programming (IP) problem. Typically, IP for large-size problems (e.g., thousands of database objects and hundreds of storage devices in a data center) are hard to solve using standard solvers like CPLEX [Ilog 2006] due to computational complexity and the resources required. Further, the NP-hardness of the problem **P** makes it harder to obtain exact solutions. In the next section, we develop a simple heuristic algorithm that provides an acceptable approximate solution to the problem with an acceptable time complexity. Moreover, unlike CPLEX and the model-based approach where we cannot get a solution when the problem is infeasible, our heuristic algorithm will provide a solution that will balance the utilization of I/O bandwidth across storage nodes while also meeting the capacity constraint. In Section 7, we evaluate the accuracy of our heuristic.

5. A HEURISTIC ALGORITHM

In this section, we present a heuristic algorithm that provides an approximate solution to the problem **P** with an acceptable time complexity. Given the current storage configuration (i.e., assignment of database objects to individual storage nodes), the heuristic aims at finding a new storage configuration that is better suited to serve the current request load.

The psuedocode for the heuristic algorithm is shown in Table II. The algorithm takes as input the current object assignment to storage nodes c_{ij}, the current bandwidth utilization of each storage node U_i, and the current I/O bandwidth consumed due to each object r_i. The algorithm produces as output a new assignment of objects to storage nodes, namely (x_{ij}). Although this algorithm requires an existing assignment of objects to storage nodes, bootstrapping the system can still be performed by starting with a random assignment of objects to storage nodes.

Greedy heuristics are known to give a good heuristic solution for various kinds of knapsack problems [Kan et al. 1993]. Also, a greedy heuristic allows us to develop a simple algorithm that can be easily adapted by data center managers. We develop a two stage greedy heuristic algorithm. In the first stage, the algorithm is greedy towards smaller size of objects, in the second stage the algorithm is greedy towards I/O bandwidth utilization. Following this, in our algorithm, we try to move the smaller objects across storage nodes first to achieve the objective goal, before choosing to move the larger ones, that is, ones *greedy on size*. In moving the objects we first try to assign objects with higher bandwidth utilization r_j to storage nodes that have lower overall percentage bandwidth utilization $\frac{U_i}{U_i^m}$, namely those, *greedy on I/O-bandwidth utilization*.

To begin, the algorithm chooses a set S of storage nodes whose bandwidth utilizations are above the threshold set in Eq. (4), or with storage sizes that do not allow future growth of the database objects that they house (line 1 of Table II). It then creates a list of objects currently residing in the node set S such that removing these objects from S will decrease the utilization of each node in S to an acceptable level. Further, in choosing the objects to place in L, the algorithm ensures that the minimum amount of data is moved from the set S of nodes. The creation of L occurs in lines 8 through 17 of the psuedocode. Additionally, for each storage $i \in I$, the O_i keeps track of the objects have already been considered for moving into another storage from storage i. This tracking eliminates the reconsideration of the same object, and chooses other objects if the attempt to move a smaller object did not succeed in the previous iteration.

In the next phase (lines 18 through 45 of the table), the algorithm places each object in list L in a storage node such that the variance in bandwidth utilization is reduced, while considering node capacity with object size and growth requirements as specified in Eq. (2). To do so, it maintains the objects in L sorted by their bandwidth utilizations, choosing first the object n with highest utilization (line 19). It then creates a target set P of nodes that can house object n, given the node-utilization-threshold constraint according to Eq. (4) (line 20). Lines 21 through 39 address the case when P is an empty set (described to follow). Otherwise, a node with least percentage I/O-bandwidth utilization in

Table II. Dynamic Data Movement Heuristic for Load-Balancing Storage Node Accesses

Input: Storage node set (I) and database object set (J),
Current configurations (c_{ij}, U_i, s_j, and r_j)
Output: The new assignment of objects to storage nodes (x_{ij})
Constants: MAX_NUMBER

```
Begin Algorithm:
```
1: Let $S \leftarrow \{ i \mid i \in I$ and $100\frac{U_i}{U_i^m} > \bar{U} + U_{th}$
or $\sum_{j \in J} x_{ij}(s_j + Tg_j) > B_i\}$
2: Let $O_i \leftarrow$ empty, $\forall i \in I$
3: Let *prevDivergence* $\leftarrow$ MAX_NUMBER
4: Let $\bar{U} \leftarrow 100\frac{\sum_j \sum_i c_{ij} r_j}{\sum_i U_i^m}$
5: Let $currentDivergence = computeDivergence(c_{ij})$
6: Let $x_{ij} \leftarrow c_{ij} \quad \forall j \in J, i \in I$
7: While ($S \neq empty$ and $prevDivergence \neq currentDivergence$) {
8: List $L \leftarrow empty$, $y_{ij} \leftarrow x_{ij}, \quad \forall i \in I, \forall j \in J$
9: $\forall i \in S$ {
10: While ($100\frac{U_i}{U_i^m} > \bar{U} + U_{th}$ or $\sum_{j \in J} x_{ij}(s_j + Tg_j) > B_i$) {
11: Choose $j \in J$ such that $x_{ij} = 1, s_j \notin O_i$, and $s_j \leq s_a$,
$\forall a \in J$ such that $x_{ia} = 1$ and $a \notin O_i$
12: $L \leftarrow L + \{j\}$
13: $x_{ij} \leftarrow 0$
14: $O_i \leftarrow O_i + \{n\}$
15: $U_i \leftarrow U_i - r_j$
16: }
17: }
18: While ($L \neq empty$) {
19: Choose $n \in L$ such that $r_n \geq r_a, \forall a \in L$
20: Let $P \leftarrow \{ k \mid \sum_j (s_j + Tg_j)x_{kj} + s_n + Tg_n \leq B_k$,
$100\frac{U_k + r_n}{U_k^m} \leq (\bar{U} + U_{th})\}$
21: If (P is empty) {
22: Choose k such that $\frac{U_k + r_n}{U_k^m} \leq \frac{U_l + r_n}{U_l^m}, \forall l \in J$ and $n \notin O_k$
23: If ($k = null$) {
24: $L \leftarrow L - \{n\}$
25: $x_{in} \leftarrow y_{in} \quad \forall i \in I$
26: goto 18
27: }
28: If ($c_{kn} = 1$) {
29: $O_k \leftarrow O_k + \{n\}$
30: Choose $o \in I$ such that $x_{ko} = 1, o \notin O_k$, and $s_o \leq s_a$,
$\forall a \in I$ such that $x_{ka} = 1$ and $a \notin O_k$
31: If ($o \neq null$) {
32: $L \leftarrow L - \{n\} + \{o\}$
33: $x_{kn} \leftarrow 1$
34: $x_{ko} \leftarrow 0$
35: $U_k \leftarrow U_k + r_n - r_o$
36: }
37: goto 18
38: }
39: }

(*Continues*)

Table II. Dynamic Data Movement Heuristic for Load-Balancing Storage Node Accesses (*Continued*)

```
40:     Else { Choose k ∈ P such that U_k/U_k^m ≤ U_a/U_a^m, ∀ a ∈ P }
41:     L ← L − {n}
42:     x_kn ← 1
43:     U_k ← U_k + r_n
44:     O_k ← O_k + {n}
45:   }
46:   S ← { i | i ∈ I and 100 U_i/U_i^m > Ū + U_th
            or Σ_{j∈J} x_ij(s_j + T g_j) > B_i}
47:   prevDivergence ← currentDivergence
48:   currentDivergence ← computeDivergence(x_ij)
49: }
End Algorithm
```

P is chosen to house object n (line 40) and requisite bookkeeping is performed (lines 41 through 44). After all elements in L have been placed, S and the *divergence* (explained later) of the I/O-bandwidth utilization of storage nodes are recomputed. If S is found to be nonempty and the divergence of bandwidth utilization is different than in the previous iteration, this process is repeated. Otherwise the algorithm terminates with the new assignment given by x_{ij}.

To handle the case when P is empty, we choose a storage node k such that: (i) k can accomodate object n, given the size constraint; and (ii) placing object n in storage node k causes the least increase in percentage bandwidth utilization across all the storage nodes where the object n was not considered before for storage node k (line 22). If no such storage node k is found, we leave the object n where it was initially assigned at the beginning of the iteration and go back to the beginning of the while-loop to start iterating for the other objects in L. If node k currently houses object n ($c_{kn} = 1$), the algorithm removes n from list L and chooses a different object o, (which has the least size of the remaining objects in k) to place in L, and performs requisite bookkeeping (lines 28 through 38). To avoid reconsideration of n in the future, it puts the object n into the list O_k of storage node k.

Computing divergence. The algorithm presented in Figure 2 computes the total divergence of I/O-bandwidth utilization across all storage nodes for which the percentage I/O-bandwidth utilization exceeds the average percentage utilization by more than the utilization threshold U_{th}. We use this value to observe how the overutilization of storage bandwidth decreases in each iteration of the algorithm.

A key feature of the proposed algorithm is that it obtains a solution even in the case where it is infeasible to achieve an allocation scheme based on the model. In case the algorithm does not find a feasible solution wherein the percentage of I/O-bandwidth utilization of all storage nodes is below that threshold value, the algorithm terminates when the divergences in storage utilization in two consecutive iterations remain unchanged. Thereby, in cases when the percentage utilization of I/O bandwidth for each storage node cannot be reduced to satisfy the utilization bound of Eq. (4) and to meet the capacity constraint of

Input: Current configurations x_{ij}
Output: *divergence*

```
Begin Algorithm:
1:  Let Ū ← 100 (Σ_j Σ_i c_ij r_j) / (Σ_i U_i^m)
2:  Let divergence ← 0
3:  ∀i ∈ I {
4:      If( Ū + U_th < 100 (Σ_j x_ij r_j) / U_i^m ) {
5:          divergence ← divergence + Σ_j x_ij r_j − ((Ū+U_th) U_i^m) / 100
6:      }
7:  }
End Algorithm
```

Fig. 2. Computing the divergence between successive solutions of the heuristic.

Eq. (2), the heuristic still provides a solution that reduces the overutilization of I/O bandwidth (i.e., divergence) across the storage nodes.

Heuristic complexity. Without exploring every detail of the algorithm, we provide only a synopsis of the complexity calculation here, focusing on the dominant computations in the algorithm. The outer loop (line 7) is repeated at most C times, a positive integer value that depends on the convergence rate of the algorithm. We demonstrate in Section 7 that typical values of C range within 5 to 6 for a few thousands of objects and about a hundred storage nodes. The outer loop at line 9 will be executed in $O(|I|)$ time. The loop at line 10 is executed on average $\frac{|J|}{|I|}$ times (average number of objects per storage). Line 11 can be executed in $O(log(\frac{|J|}{|I|}))$. Thus the average order of complexity for the loop at line 10 for each iteration of the loop at line 7 is $O(|I|\frac{|J|}{|I|}log(\frac{|J|}{|I|}))$. The inner loop (line 18) is repeated on average $\frac{|J|}{|I|}$ times. Further, for each iteration of the inner loop (line 18), line 19 has the average time complexity of $O(\frac{J}{I})$. Line 20 has the complexity of $O(log(|I|))$. One of line 22 or line 40 is executed in each iteration of the inner loop (line 18), both of which have complexity of $O(|I|)$. Line 30 is executed in the small percentage of cases where $c_{kn} = 1$. Let us assume that it is executed for a δ-fraction of the total execution of the outer loop (line 18). Line 30 has a complexity of $O(log(\frac{|J|}{|I|}))$. Line 46, which belongs to the outer loop, has time complexity of $O(|I|)$. The total average time complexity of the algorithm is thus computed to be $O(C \times (|I|\frac{|J|}{|I|}log(\frac{|J|}{|I|}) + \frac{|J|}{|I|}(\frac{|J|}{|I|} + |I| + \delta log(\frac{|J|}{|I|}) + |I|))$. Considering that C is a small number (typically around 5 to 6, based on experimental data) and that $|J| > |I|$ (i.e., the number of database objects is larger than number of storage nodes, a typical case), we conclude that the total time complexity of the heuristic algorithm is $O(|J|log(\frac{|J|}{|I|} + (\frac{|J|}{|I|})^2)$. Thus, our heuristic algorithm is low-order polynomial and can be run quickly for a large number of database objects and storage nodes.

6. MONITORING USAGE PATTERNS

In this section we describe how STORM collects the usage statistics of database objects and their storage consumptions from commercially available databases.

These statistics capture the usage behavior of standard classes of database objects (tables and indices) found in DBMSs. Note that these usage patterns are very often the objects of enquiry for DBAs becuase they lend significant insight into how database objects should be placed in various storage devices across the network. Specifically, Storm collects three *base usage statistics* automatically, namely:

1. frequency of access of database objects;
2. average data-access size of queries for a database object; and
3. storage consumption and growth estimate of objects.

In the Storm architecture, as presented in Figure 1, the DBMON and STMON are profiling modules for the database and storage system, respectively. Both processes are configured with a list of database servers (encompassing all the database server clusters) to be monitored. For monitoring database access patterns, the DBMON submits a set of monitoring queries to the database servers, and receives the results. The DBMON then processes these results to yield the monitoring information of interest. The STMON also queries the database servers and storage devices to obtain dynamic information about space usage for individual database objects by each database server, and static information such as the storage capacity of individual devices.

The aforesaid modules are designed so that they are nonintrusive. The data queried in the monitoring process is not application data, but rather system metadata. As a result, not only is the total volume of data collected small, but also there is minimal contention with application data access. Further, the query interval is typically large and also configurable; it can be made sensitive to database resource availability (e.g., can be done during off-peak hours or service down-time). Finally, once the query results have been received, all other processing takes place outside the database process.

Next, we describe specific practical techniques to gather the previously described information, which apply to most commercial off-the-shelf (COTS) databases. We specifically describe our approach for Oracle and SQL-Server databases, but note that our scheme can easily be extended to other COTS databases such as MySQL, DB2, and Sybase.

6.1 Details of Data Gathering Technology

We now describe a specific set of SQL queries that can be used by DBMON and STMON to derive usage statistics of database objects and their storage consumption. This approach is applicable for all COTS databases; however, the specific SQL queries needed to gather are dependent on the metadata architecture of each vendor's DBMS.

6.1.1 *Database-Object Access Patterns.* Table III lists specific queries that DBMON uses to obtain the frequency of accesses of database *table* and *index* objects. From the output of Query Q1, DBMON can generate a list of tables (e.g., CUSTOMER) and columns, prefaced with their associated table names (e.g., CUSTOMER.ZIPCODE) in the database. We call these two lists the *TableList*

Table III. Database Object-Usage Monitoring Queries

	DBMS	Query
Q1	Oracle	SELECT TABLE_NAME COLUMN_NAME FROM DBA_ALL_TABLES
Q1	SQL-Server	SELECT DATABASE_NAME.DBO.SYSOBJECTS.NAME TABLE_NAME, DATABASE_NAME.DBO.SYSCOLUMNS.NAME, COLUMN_NAME FROM DATABASE_NAME.DBO.SYSOBJECTS, DATABASE_NAME.DBO.SYSCOLUMNS WHERE DATABASE_NAME.DBO.SYSOBJECTS.ID=DATABASE_NAME.DBO.SYSCOLUMNS.ID AND DATABASE_NAME.DBO.SYSOBJECTS.XTYPE = 'U'
Q2	Oracle	SELECT c.INDEX_NAME, c.COLUMN_NAME, c.TABLE_NAME i.NUM_ROWS FROM DBA_IND_COLUMNS c, DBA_INDEXES i WHERE c.INDEX_NAME=i.INDEX_NAME
Q2	SQL Server	SELECT I.NAME INDEX_NAME, I.INDID, O.NAME TABLE_NAME FROM DATABASE_NAME.DBO.SYSINDEXES I, DATABASE_NAME.DBO.SYSOBJECTS O WHERE I.ID = O.ID AND INDID > 0 AND INDID < 255 AND O.TYPE = 'U' ORDER BY O.NAME INDEX_COL (TABLE_NAME , I.INDID , key_id) /* returns the column name where key_id is the ID of the key*/
Q3	Oracle	SELECT DISTINCT s.SQL_TEXT SQL_TEXT, s.EXECUTIONS EXECUTION_COUNT FROM V$SQL s
Q3	SQL Server	SELECT SQL SQL_TEXT, USECOUNTS EXECUTION_COUNT FROM MASTER.DBO.SYSCACHEOBJECTS, MASTER.DBO.SYSDATABASES WHERE UID >1 AND ((CACHEOBJTYPE= 'EXECUTABLE PLAN' AND OBJTYPE='PREPARED') OR (CACHEOBJTYPE='EXECUTABLE PLAN' AND OBJTYPE='PROC' AND SQL NOT LIKE 'SP_%')) AND SYSCACHEOBJECTS.DBID = SYSDATABASES.DBID AND MASTER.DBO.SYSCACHEOBJECTS.DBID=DB_ID('DATABASE_NAME')

and *ColumnList*, respectively. From the output of Query Q2, DBMON generates a list of indices, where each index is identified by the table and column it indexes (e.g., CUSTOMER.ZIPCODE_INDEX). We call this the *IndexList*.

Q3 returns a list of SQL statements executed on the database, and each statement's execution count. We call this list the *ExecutedStatementList*. Specifically, each item in this list represents a single statement, and contains the StatmementText (e.g., "select distinct ZIPCODE from CUSTOMER") and an integer *StatmentExecCount*. For Q3, we note that some consideration must be given to the query submission interval. The tables in Oracle and SQL-Server that store executed statements and execution counts are part of the DBMS's caching infrastructure, and are flushed at an administrator-determined interval. The interval between issuing Q3 queries must be smaller than this interval.

DBMON can determine database-object usage statistics by *aggregating access counts*. The pseudocode in Algorithm 1 describes this aggregation process.

Algorithm 1. Aggregate Usage Statistics

```
1: for all tables t in TableList do
2:    Create a count_t variable, and initialize it to 0
3:    Create a RetDataSize_t variable and initialize it to 0
4: for all columns c in ColumnList do
5:    Create a countIndexUse_c variable, and initialize it to 0
6:    Create a countIndexNeeded_c variable, and initialize it to 0
7: Create an empty restrictionUseHashTable
8: for all  statement s in ExecutedStatementList do
```

```
 9:    Create three empty lists, tableAccessList, columnAccessList, and
       columnValueList
10: Parse s's StatementText. Add all table names accessed in the WHERE clause
    to the tableAccessList, and all column names accessed in the WHERE clause
    to the columnAccessList. Add column name-value pairs for all column values
    accessed in the WHERE clause to the columnValueList
11: Based on the range values and selectivity analysis ([Aoki 1999; An et al. 2003;
    Aboulnaga and Chaudhuri 1999] DBMON estimates the size of return data
    set for each table in tableAccessList
12: Update RetDataSize_t based on computed return data size and StatementExecCount
13: for all tables in the tableAccessList do
14:    Increment count_t by s's StatementExecCount
15: for all columns in the columnList do
16:    if the column name matches an index in the IndexList then
17:      Increment countIndexUsed_c by s's StatementExecCount
18:    else
19:      Increment countIndexNeeded_c by s's StatementExecCount
20: for all items in the columnValueList do
21:    if the column-value pair does not exist in the restrictionUseHashTable then
22:      Create a new hash element, with the column-value pair as the key, and s's
         StatementExecCount as the value
23:    else
24:      Increment the hash element value for the column-value pair by s's
         StatementExecCount
25: Return the restrictionUseHashTable, count_t for all tables, as well as
    restrictionUse_c, countIndexUsed_c and countIndexNeeded_c for all columns
```

Lines 1 to 6 create a set of aggregation structures: the set of $count_t$ variables store the number of accesses to each table, and the sets of column variables $countIndexUsed_c$ and $countIndexNeeded_c$ represent the number of accesses to columns for restriction purposes, where an index would be used if available. The *restrictionUseHashTable* structure stores restriction uses for specific column values and $RetDataSize_t$ represents the total amount of data returned from a table on a set of queries. Lines 8 to 9 parse the query into tables, restriction columns, and column-values accessed, while lines 12 to 23 update the $count_t$, $countIndexUsed_c$, $countIndexNeeded_c$, and *restrictionUseHashTable* with the execution counts for each query. With this information, DBMON is now prepared to compute the usage statistics that will be required by our approach (as described in Section 5) for the *decision maker* of Storm.

Frequency of access of database objects. The $count_t$ and $countIndexUsed_c$ values give us the usage statistics for database objects, tables and indices.

Average data return size of queries for a database object. Combining the $count_t$ along with $RetDataSize_t$, DBMON can accurately compute r_j, the average bytes/sec retrieved from the object (table).

Cache considerations. Modern database systems cache frequently accessed objects in memory. This introduces an error into our original calculations where we assume that all data retrieved for query computations is from storage devices. The effect of cache accesses can be taken into account in the exact computation of the usage statistics in the following way. Table IV shows the query and system stored procedures in Oracle and SQL-Servers, respectively, that provide specific statistics about cache hits. The Oracle query returns a list of queries

Table IV. Cache Usage Queries

DBMS	Query
Oracle	SELECT executions, buffer_gets, disk_reads, first_load_time, sql_text FROM v$sqlarea ORDER BY disk_reads
SQL-Server	DBCC MEMUSAGE

Table V. Storage Consumption Queries

	DBMS	Query Description	Query
Q4a	Oracle	Table-space consumption	SELECT df.TABLESPACE_NAME, SUM (df.BYTES) TOTAL_SPACE, SUM(fs.BYTES) FREE_SPACE, ROUND(((NVL(SUM (fs.BYTES),0)/SUM(df.BYTES))*100),2) PCT_FREE FROM DBA_FREE_SPACE fs, DBA_DATA_FILES df WHERE df.TABLESPACE_NAME = fs.TABLESPACE_NAME (+) GROUP BY df.TABLESPACE_NAME ORDER BY df.TABLESPACE_NAME
Q4b	Oracle	Physical files associated with a Tablespace	SELECT FILE_NAME, TABLESPACE_NAME FROM DBA_DATA_FILES WHERE STATUS='AVAILABLE';
Q4	SQL Server	Physical files associated with a Filegroup	SELECT FILEGROUP_NAME(GROUPID), GROUP_NAME, FILESIZE TOTAL_SPACE, FILEMAXSIZE-FILESIZE FREE_SPACE, FILEMAXSIZE, GROWTH = (CASE SYSFILES.STATUS & 0X100000 WHEN 0X100000 THEN CONVERT(NVARCHAR(3), GROWTH) + N'%' ELSE CONVERT(NVARCHAR(15), GROWTH * 8) + N' KB' END), NAME LOGICAL_FILENAME ,FILENAME FROM DATABASENAME.DBO.SYSFILES WHERE GROUPID <> 0
Q5	Oracle	Size of index and tables	SELECT sum(bytes)/1048576 Megs, segment_name FROM user_extents WHERE segment_name = 'object_name' GROUP BY segment_name
Q5	SQL Server	Size of index and tables	EXEC sp_spaceused 'table_name' @updateusage = 'true';

that required disk access, along with the number of such disk accesses. The SQL-Server stored procedure gives information about queries that are being served from the buffer cache, that is, where disk access is not required. This information can then be used by DBMON, along with previously obtained information, to accurately compute the average bytes/sec retrieved r_j from various database objects, taking into account the effect of the database buffer cache.

6.1.2 *Storage Consumption Patterns.* Table V lists queries used by STMON to obtain information about object storage consumption and storage consumption growth (base usage pattern **3**). For Oracle, STMON generates a list of table spaces, where each table space is associated with the total space

for the table space, free space, and the percentage of free space from the output of Query Q4a in Table V. We can add the filename(s) associated with the table space from the output of Query Q4b. For SQL-Server, all these data elements can be obtained for filegroups from Q4 for SQL-Server. Note that the expression in parentheses (CASE...END) in this query is an embedded stored procedure designed to convert binary-format numbers into integer format. With this information STMON can compute c_{ij} (i.e., the current assignment of a database object to storage nodes).

Query Q5 gives us the size of database-object tables and indices s_j in Oracle and SQL-Server. Taking a series of these values over time provides data points which STMON uses to forecast the growth of table size over time g_j.

7. SIMULATION RESULTS

In this section, we demonstrate the efficacy of our heuristic algorithm in terms of two key metrics: (i) accuracy, and (ii) convergence. We do so, using simulation techniques.

7.1 Accuracy of Heuristic Algorithm

To evaluate the accuracy of the heuristic, we compare the data movement required by the heuristic solution to that required for the solution obtained by solving the model **P** using CPLEX [Ilog 2006]. Due to the NP-hardness of **P**, large-size problems (e.g., 100 storages and 3000 objects) cannot be solved using CPLEX within an acceptable time bound. We therefore resort to the alternate approach of calculating the accuracy using a lower bound of the problem **P**. We obtain the lower-bound solution of the problem **P** by LP-relaxation (relaxing the binary constraint of the variable x_{ij} and declaring it as a variable within {0,1} range) and solving the LP-relaxed version of the problem **P**. We compute the percentage gap between the data movement obtained by the heuristic and this lower bound as follows.

$$PercentageGap = \frac{Heuristic - LowerBound}{LowerBound} \times 100$$

We generated feasible problems by varying the size of database objects j uniformly within 1–100MB. We varied the r_j for database objects following a Zipfian distribution (following the typical notion that 20% of database objects are accessed in 80% of the cases) within 10–1000 bytes/sec. We varied the growth rate of database objects uniformly between 0–1000 KBytes/days. For baseline, we set the number of storage nodes to 100 and the number of objects to 200. Then, fixing storage nodes to baseline value (100), we varied the number of objects (500, 1000, 1500, 2000, 2500, and 3000). Similarly, keeping the number of objects to the baseline value of 2000 we varied the number of storage nodes (50, 75, 100, 125, 150, and 175). We thus obtained 11 cases in total. The value of T is kept constant at 15 days (2 weeks) and the threshold value of utilization U_{th} is kept at 5%. For each of these cases, we generated three problem instances by varying the parameters as described earlier. In each case, the value of storage node sizes B_i and maximum bandwidth U_i^m are generated randomly within an upper bound and lower bound computed so that feasible solutions of the model

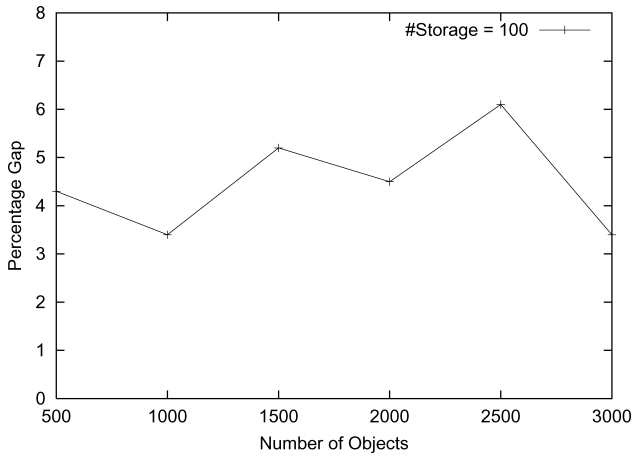

Fig. 3. Varying numbers of objects.

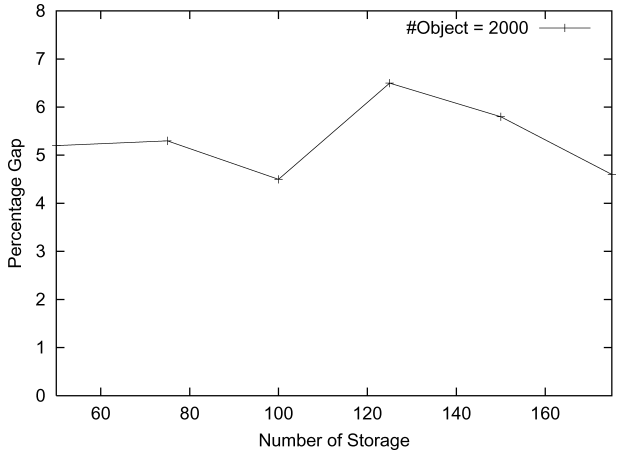

Fig. 4. Varying numbers of storage nodes.

exist. We then compute the average percentage gap in data movement for each case.

Variation of the *PercentageGap* with varying numbers of objects and storage nodes are shown in Figures 3 and 4, respectively. In both cases, the gap remains within 7% of the lower bound. This also implies that in cases where a feasible solution exists, the heuristic solution lies within 7% of the optimal solution. Note, however, that here we compare against the lower-bound solution, which may be worse than the optimal. So, the actual gap may be lower than those shown in the figures.

7.2 Convergence of Heuristic Algorithm

To demonstrate the convergence of the heuristic, we plot in Figure 5 the value of *currentDivergence* (as computed in the Figure 2) in each iteration. Note how the *currentDivergence* reduces in each successive iteration. The graph shows

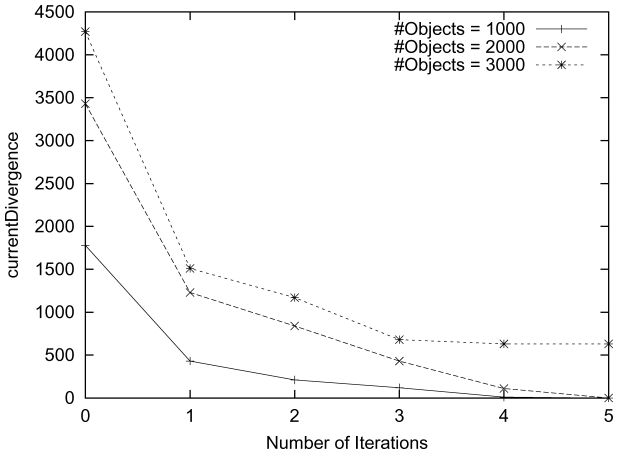

Fig. 5. Convergence of heuristic algorithm.

this *currentDivergence* for three different values of number of objects 1000, 2000, and 3000; the number of storage nodes is kept constant at 50. The values of other parameters are generated as described in Section 7.1. The values of storage node sizes are generated based on feasibility of the solution. The value of maximum bandwidth of storage nodes is generated as a random number between 15 and 25 MBytes/sec in each of these three cases.

As can be seen from Figure 5, the algorithm converges in all three cases within at most 5 iterations. In the case of 1000 and 2000 objects, the algorithm produces a feasible solution with a final divergence of zero. In the case of 3000 objects, the algorithm is unable to produce any feasible solution. However, unlike a model-based CPLEX approach that does not produce any solution in the case of infeasibility, the heuristic actually reduces the divergence from 4500 to about 600 and stabilizes before exiting within 5 iterations. Thus, our heuristic not only generates a solution that is near optimal, but also does so within a very small number of iterations. Further, the efficiency of the heuristic is excellent. The heuristic required approximately 5 sec on an average for a run with 3000 objects and 175 storage nodes on a Pentium 2.4 GHz processor. We therefore believe that the heuristic is practical and can easily be used for online database storage management in a data center environment.

8. EXPERIMENTAL RESULTS

In this section, we evaluate a prototype implementation of Storm in its ability to optimize the bandwidth utilization of multiple storage nodes. We evaluate both the improvement in utilization of individual drives (by balancing such utilization equally across the nodes) as well as the overall performance of the database system in terms of transactions-per-minute-count (TPMC).

For the database server, we used PostgreSQL 8.2 [PostgreSQL Global Development Group 2007] on Linux (2.6.18.1 kernel), running on a Pentium 4 2GHz machine with 1GB RAM. The machine had five disk drives: `hda`, `sda`, `sdb`, `sdc`, and `sdd`. We emulated multiple storage nodes within this single machine,

using each disk drive as an independent storage node. The operating system was installed on `hda`, while the other four drives were used for storing the database objects of the PostgreSQL database.

For the workload, we used the TPC-C benchmark [Transaction Processing Council 2003], a benchmark that mimics an online transaction processing workload, running on a remote client machine. We used an open source implementation of the benchmark available at `http://sourceforge.net/projects/benchmarksql/`. The TPC-C benchmark uses nine tables: Customer, Warehouse, District, History, Neworder, Order, Orderline, Item and Stock. In addition to primary key indices there are two additional indices on Customer and Order tables. The PostgreSQL database also uses base and pg_xlog directories for its internal operation. The base directory contains the metadata information about the database and the pg_xlog directory contains transaction logs for transaction recovery in the database. The read and write operations to these two directories were significant compared to the operations on tables and indices related to TPC-C. In using Storm, we considered the base and pg_xlog directories as database objects as well.

The input to the benchmark is comprised of the number of terminals, number of warehouses, and the execution duration. The number of warehouses was set to 100 to create a database of size 10GB. The duration of the experiment was set to 30 minutes for each run. The number of terminals simulates parallel execution of the online transactions from multiple clients, and was varied from 1 to 96. The benchmark reports the TPMC for each experiment. We also implemented a Linux kernel module that reports the average bandwidth utilization of each disk drive over the duration of the experiment by intercepting the block I/O issued to each drive.

To demonstrate the advantages of the Storm approach, we compared three different allocations of database objects to storage nodes. In the first, all database objects were located in a single drive, representing a skewed allocation, which we denote as "Single-Disk Allocation." Next, the database objects were distributed manually across the four drives to approximately balance storage-space utilization. With this setup, we attempt to mimic the behavior of a database administrator who does not have access to per-object disk utilization. We denote this as "Initial Allocation." Next, assuming the initial allocation as the starting state, we ran the Storm algorithm to determine an optimized allocation of database objects to the four drives. We denote this as "Storm Allocation." In each of the three cases, we ran the benchmark for 30 minutes while also varying the number of terminals (to vary the system load). In each case we noted the transactions-per-minute-count (TPMC) reported by the benchmark and the I/O-bandwidth utilization (KBytes/sec) for each of the drives.

8.1 Transactions Per Minute

Figure 6 demonstrates how TPMC varies with the number of terminals in the benchmark. As expected, the single-disk allocation performance is substantially worse than both the initial allocation and Storm allocation. This underscores the benefit of distributing database objects across multiple storage nodes for parallelized I/O.

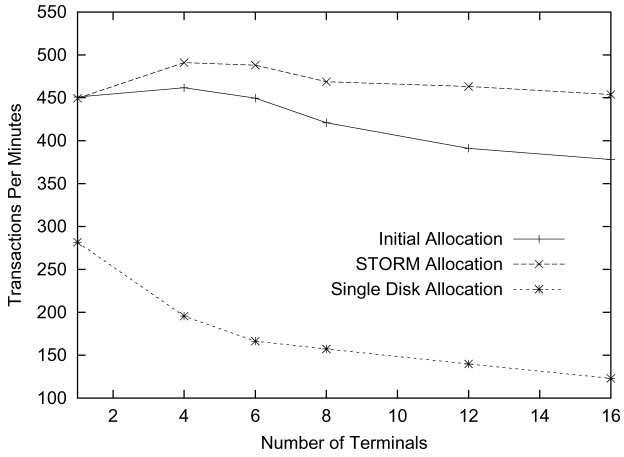

Fig. 6. Impact of increased query parallelism on TPMC.

In case of both Storm allocation and initial allocation, as we increase the number of terminals, after an initial increase in the TPMC, it starts to degrade. When the number of terminals is increased from 1 to 4, the incoming rate of transactions goes up and this results in increased overall TPMC. However, further increases in the number of terminals results in a decreased TPMC. This behavior is an artifact of the benchmark, whose transactions progressively increase the size of the database over the benchmark's execution. With a larger number of terminals, the rate at which the size of the database increases is also greater. The resulting phenomenon is that successive transactions (even of the same kind) take longer to complete as time progresses, thereby reducing the TPMC. The acceleration of this phenomenon is greater for a larger number of terminals. Interestingly, as we shall see shortly, the average disk utilization for both Storm allocation and initial allocation continues to increase beyond no. terminals = 4, and maximizes only at no. terminals = 64, when an upper bound on the bandwidth utilization for the bottleneck drive(s) is reached.

Comparing the performance of the initial allocation and Storm allocation, the TPMC is same for both at no. terminals = 1. As the number of terminal increases, we see improved performance of the Storm allocation case compared to the initial allocation. At no. terminals = 16, the TPMC in case of Storm allocation is 453 compared to 378 in initial allocation, which is about, 20% improvement. Note, in fact, that these percentages are underestimated due to the benchmark artifact that we described earlier.

As the number of terminals is further increased, due to the reasons explained before, the TPMC decreases further in all three cases. However, because the TPMC values at higher numbers of terminals do not give much insight, we do not present this data in Figure 6.

8.2 Storage Bandwidth Utilization

The primary reason for the improved performance in the Storm case is because of the effective utilization of I/O bandwidth across storage nodes. We

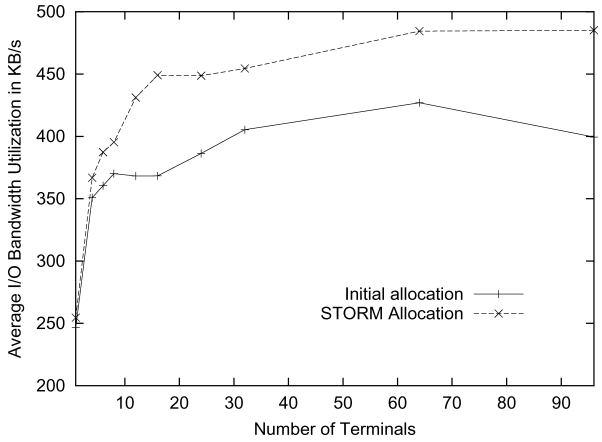

Fig. 7. Average I/O-bandwidth utilization.

demonstrate this with two experiments that measure: (i) the average I/O-bandwidth utilization and (ii) the percentage standard deviation in I/O-bandwidth utilization across storage nodes. The latter measurement aims at demonstrating how Storm is more effective in avoiding a bottleneck at a single storage node.

In Figure 7, we present the average I/O-bandwidth utilization (KBytes/sec) across four data drives for both the initial allocation and Storm allocation. With increasing numbers of terminals, the average I/O-bandwidth utilization increases for both the initial allocation and Storm allocation. In both case, the average bandwidth utilization maximizes at no. terminals = 64. However, at no. terminals = 96, the average bandwidth utilization decreases in the case of initial allocation whereas in the case of Storm allocation remains constant. This is because the Storm allocation algorithm is able to distribute the I/O-bandwidth utilization across storage nodes more uniformly than the initial allocation.

At all values of the terminals, the average I/O-bandwidth utilization across storage nodes is higher in the case of Storm allocation. At no. terminal = 64, the average I/O-bandwidth utilization in the case of Storm allocation is 14% higher and at no. terminals = 96, it is 22% higher. Thus, Storm allocation algorithm results in more efficient database-object distribution across various drives increasing the overall I/O-bandwidth utilization across storage nodes. We explore this aspect further by examining the standard deviation in I/O-bandwidth utilization across storage nodes.

Figure 8 presents the percentage of standard deviation of I/O-bandwidth utilization (KBytes/sec) across four data drives for both initial allocation and the Storm allocation. In both cases, the percentage of standard deviation decreases initially with the increasing number of terminals in the benchmark. Upon increasing the number of terminals, an increasing number of execution threads issuing varying transactions simultaneously makes the access to various database objects more uniform, restricted only by the distribution of object utilizations due to the benchmark. In both allocations, after initial rapid

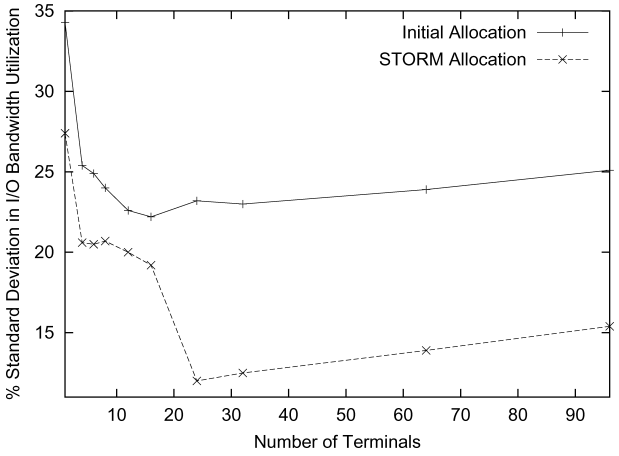

Fig. 8. Percentage of standard deviation in I/O-bandwidth utilization.

decrease, the standard deviation remains more or less constant. At this point the standard deviation is due more to the imbalance in I/O-bandwidth utilization across the four drives. At all values of no. of terminals, the standard deviation is less in case of Storm allocation. At no. terminals = 4, the standard deviation in Storm allocation is 20% less than that of the initial allocation. At no. terminals = 16, the standard deviation in Storm allocation is 15% less, and at no. terminals = 24, the standard deviation is 50% less. Thus the Storm allocation algorithm results in more uniform I/O-bandwidth utilization across storage nodes, which in turn results in improved end-to-end performance of the database system. Moreover, Storm allocation does much better at higher levels of parallelism due to greater pronunciation of the I/O bottlenecks.

8.3 Summary

We experimentally demonstrated herein that the Storm approach in databases object allocation across multiple disks results in about 22% improvement in disk I/O-bandwidth utilization, which in turn improves the overall transactions per minute of the database by 20%. We want to note that due to the limited resources available in an academic environment, we had to resort to a very small environment with just four storage drives and 13 database objects (9 tables, 2 indices, and 2 database directories). We demonstrated that even within this small setup (which is actually a very popular benchmark for database systems), we have been able to improve the system performance by 22%. In large systems with thousands of database objects (e.g., a PeopleSoft Enterprise system has 22,000 tables and an equivalent number of indices) and a few tens of storage nodes holding several hundreds GBytes of data, the complexity of distribution of database objects across storage nodes will be much more complex. So, in large environments the manual effort of distributing database objects across storage nodes will lead to very suboptimal performance. In such scenarios, the impact of our Storm approach on the overall performance of the system is expected to be substantially greater, reflecting the simulation results we presented in Section 7.

9. CONCLUSION

In this article we presented Storm, an automated approach that can guide effective the utilization of the storage nodes used for database storage in a data center environment. We developed a math programming model of the problem and showed that the the problem is NP-hard. For data center managers, we developed a simple greedy heuristic that quickly provides approximate solutions. By simulation study we have shown that the greedy heuristic generates a solution for a feasible problem that lies within 7% of the optimal solution. Further, even in cases where the actual formulation of the problem does not allow feasible solutions, the heuristic is still effective in significantly reducing the imbalance in bandwidth utilization across storage nodes. The time complexity of the heuristic algorithm is low-order polynomial, making it an efficient and practical solution for large numbers of database objects (several thousands) and storage nodes (few tens). Lastly, we demonstrated by experimental study that we have been able to improve the performance of a database benchmark by 22% with the Storm approach.

Currently, our system operates at the granularity of a database object such as a table or an index. Therefore, we cannot accommodate splitting of tables or indexes across multiple storage nodes. In the future, we intend to incorporate techniques that can derive access and usage data information at the database block level, whereby our technique can distribute various blocks of the same database object (table or index) in multiple storage nodes. This approach will achieve better distribution of I/O bandwidth in case of highly-skewed intra-object utilization.

As another future research agenda, we intend to extend our work to other storage systems such as mail systems and file systems. For file systems, we can look at "files" as objects. For email systems, we can look at mailbox folders (both user-mail boxes and incoming message folders), as well as auxilliary entities such as user address-books, calendars, etc. (if these are used) as objects. Similarly, in other storage systems, once we have identified the objects and mechanisms to derive usage parameters of these objects (such as storage space, I/O bandwidth utilizations due to each of these objects), we can apply our heuristic algorithm (Table II) *as-is* to determine the desired allocation scheme. In future, we intend to compare the performance derived by our algorithm with that of OS level block allocation schemes across multiple storage nodes in the case of file and mail systems.

REFERENCES

ABOULNAGA, A. AND CHAUDHURI, S. 1999. Self-Tuning histograms: Building histograms without looking at data. In *Proceedings of the International Conference on Management of Database ACM SIGMOD*. Philadelphia, PA, 181–912.

ALLEN, N. 2001. Don't waste your storage dollars: What you need to know. *Res. Note*, Gartner Group.

AN, N., JIN, J., AND SIVASUBRAMANIAM, A. 2003. Algorithms for index-assisted selectivity estimation. *IEEE Trans. Knowl. Data Eng. 15*, 2, 305–323.

AOKI, P. 1999. Toward an accurate analysis of range queries on spatial data. *Proceedings of the 15th International Conference on Data Engineering*, 258.

BMC SOFTWARE. 2005. Capacity management and provisioning. www.bmc.com.

CAPPANERA, P. AND TRUBIAN, M. 2005. A local search based heuristic for the demand constrained multidimensional knapsack problem. *INFORMS J. Comput. 17*, 82–98.

CHU, P. AND BEASLEY, J. 1998. A genetic algorithm for the multidimensional knapsack problem. *J. Heuristics 4*, 63–86.

COMPUTER ASSOCIATES. 2005. Storage management. www.ca.com/products.

FENG, Y. AND ZHANG, Y.-Y. 2005. Virtual disk reconfiguration with performance guarantees in shared storage environment. In *Proceedings of the 3rd International Conference on Information Technology and Applications*, 69–74.

FURTADO, P. 2004. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. ACM Press, New York, NY, USA, 23–30.

GANGER, G. R., WORTHINGTON, B. L., HOU, R. Y., AND PATT, Y. N. 1993. Disk subsystem load balancing: Disk striping vs. conventional data placement. In *Proceedings of the International Conference on System Sciences*.

HUA, K. A. AND LEE, C. 1990. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, 493–506.

IBM 2006. Storage area network (SAN). http://www-03.ibm.com/servers/storage/san/.

ILOG 2006. ILOG CPLEX World's leading mathematical programming optimizers. http://www.ilog.com/products/cplex/.

KAN, A. H. G. R., STOUGIE, L., AND VERCELLIS, C. 1993. A class of generalized greedy algorithms for the multi-knapsack problemm. *Discr. Appl. Math. 42*, 279–290.

KARL NAGEL CORPORATION 2006. Sarbanes-Oxley. http://www.sarbanes-oxley.com/.

KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. *IEEE Comput. 36*, 1 (Jan.), 41–50.

KHULLER, S., KIM, Y., AND WAN, Y. 2003. Algorithms for data migration with cloning. In *Proceedings of the 22nd ACM Conference on Principles of Database Systems*.

KWAN, T. T., MCCRATH, R., AND REED, D. A. 1995. NCSA's World Wide Web server: Design and performance. *IEEE Comput. 28*, 11, 68–74.

LAMB, E. 2001. Hardware spending spatters. *Red Herring*, 32–33.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 84–92.

LU, C., ALVAREZ, G. A., AND WILKES, J. 2002. Aqueduct: Online data migration with performance guarantees. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 219–230.

MCDATA CORP. 2006. Storage network extension and routing. http://www.mcdata.com/products/hardware/srouter/index.html.

MEHTA, M. AND DEWITT, D. J. 1997. Data placement in shared-nothing parallel database systems. *The VLDB J. 6*, 1, 53–72.

MENON, J., PEASE, D. A., REES, R., DUYANOVICH, L., AND HILLSBERG, B. 2003. IBM storage tank—A heterogeneous scalable SAN file system. *IBM Syst. J. 42*, 2.

MESNIER, M., GANGER, G. R., AND RIEDEL, E. 2003. Object-Based storage. *IEEE Commun. Mag.*

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 109–116.

POSTGRESQL GLOBAL DEVELOPMENT GROUP. 2007. Postgresql 8.2. http://www.postgresql.org/.

QIAO, L., IYER, B. R., AGRAWAL, D., AND ABBADI, A. E. 2006. Automated storage management with QoS guarantees. In *Proceedings of the International Conference on Data Engineering*.

SIVATHANU, M., BAIRAVASUNDARAM, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2005. Database-Aware semantically-smart storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

STONEBRAKER, M., AOKI, P., DEVINE, R., LITWIN, W., AND OLSON, M. 1994. Mariposa: A new architecture for distributed data. In *Proceedings of the 10th International Conference on Data Engineering*, 54–65.

TRANSACTION PROCESSING COUNCIL. 2003. Automatic storage management technical overview: An oracle white paper. Oracle Technology Network (http://www.oracle.com/technology/).

TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC). 2006. TPC benchmark C standard specification revision 5.8.0. Oracle Technology Network (http://www.oracle.com/technology/).

VERITAS. 2005. Storage and server automation. http://www.symantec.com/Products/enterprise.

WU, C. AND BURNS, R. 2005. Tunable randomization for load management in shared-disk clusters. *ACM Trans. Storage 1*, 1 (Feb.), 108–131.

ZHANG, G., SHU, J., XUE, W., AND ZHENG, W. 2007. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Trans. Storage 3*, 1 (Mar.).

Received July 2007; revised October 2007; accepted December 2007