An Object Placement Advisor for DB2 Using Solid State Storage

Mustafa Canim
University of Texas at Dallas
Richardson, TX, USA
canim@utdallas.edu

George A. Mihaila
IBM Watson Research Center
Hawthorne, NY, USA
mihaila@us.ibm.com

Bishwaranjan Bhattacharjee
IBM Watson Research Center
Hawthorne, NY 10598
bhatta@us.ibm.com

Kenneth A. Ross
Columbia University
New York, NY, USA
kar@cs.columbia.edu

Christian A. Lang
IBM Watson Research Center
Hawthorne, NY 10598
langc@us.ibm.com

ABSTRACT

Solid state disks (SSDs) provide much faster random access to data compared to conventional hard disk drives. Therefore, the response time of a database engine could be improved by moving the objects that are frequently accessed in a random fashion to the SSD. Considering the price and limited storage capacity of solid state disks, the database administrator needs to determine which objects (tables, indexes, materialized views, etc.), if placed on the SSD, would most improve the performance of the system. In this paper we propose a tool called "Object Placement Advisor" for making a wise decision for the object placement problem. By collecting profile inputs from workload runs, the advisor utility provides a list of objects to be placed on the SSD by applying heuristics like the greedy knapsack technique or dynamic programming. To show that the proposed approach is effective in conventional database management systems, we have conducted experiments on IBM DB2 with queries and schemas based on the TPC-H and TPC-C benchmarks. The results indicate that using a relatively small amount of SSD storage, the response time of the system can be reduced significantly by considering the recommendation of the advisor.

1. INTRODUCTION

Solid state disks (SSDs) have been introduced by a number of vendors in the last few years. SSDs provide persistent data storage using a form of solid-state flash memory. To the operating system, they appear just like a conventional disk device. Unlike hard disks, which have a mechanical latency associated with seeks and rotation, there is a very small overhead for random access to data relative to sequential access. As a result, a solid state disk can support one to two orders of magnitude more random I/O operations per second than a hard disk [8, 17].

Currently, SSDs are more expensive than traditional hard disks when measured in gigabytes per dollar, or in terms of sequential I/Os per second per dollar. However, SSDs provide better value than hard disks if one measures random I/Os per second per dollar [8]. Thus, SSDs provide an opportunity to improve the performance of a system if it typically incurs a lot of random I/O. At the same time, one does not typically want to put all of one's data on SSDs, because the cost per gigabyte is higher than on a hard disk. Ideally, one should put "just enough" data on SSD storage so that a large fraction of the potential random I/O savings can be realized. Beyond a certain point, the marginal gains in random I/O saving may be small, and outweighed by the increased cost of the extra SSD devices.

Consider Bob, the database administrator for a large corporation.[1] Bob is likely to need answers to the following questions:

- If I bought a single SSD device that can accommodate only part of my data, which data should be placed on it?
- Given the answer to the previous question, how much better would my performance be?
- Would it be worth the expense to purchase multiple SSD devices?
- What aspects of SSD performance should I care most about when choosing an SSD product?

If these questions are not adequately addressed, then Bob could easily make one of the following poor decisions:

- Buying a reasonable amount of SSD storage, but using it suboptimally, so that the apparent benefits are smaller than what is possible.

[1]The questions listed would also be appropriate for the administrator of any complex system with persistent storage, such an an operating system or file system, but our focus in this paper is on database systems.

VLDB '09, August 24-28, 2009, Lyon, France

- Not buying any SSD storage, because a trial-and-error approach to placing objects on SSDs did not yield significant savings, even when there were significant benefits available with an informed data placement choice.

- Incurring the cost of buying too much SSD storage, when less SSD storage would have performed just as well.

- Buying the wrong kind of SSD storage, with the aspects of SSD performance most critical to the application having not been the aspects on which the product choice is made.

These are real concerns in today's marketplace. Consider the following comment made by the editor of a market survey on the potential use of solid state disks in industry [13]:

> *The fact that over half the responders cited performance guarantees as a gating factor to buying SSDs suggests that users have seen far too many unmet promises about performance in other aspects of their IT experience... One problem is that the application speedup in practise is going to vary according to the hardware environment and application. The pure performance of the product does not tell the whole story.*

Answering Bob's questions in the context of a database system is challenging.

- Current cost-based query optimizers can take account of disk characteristics in order to estimate execution times. However, until now, disk characteristics within a single installation have typically been relatively uniform. Thus there has been little need for a database utility to consider the various options for data placement.

- Even if a query optimizer is intelligent enough to distinguish between sequential and random I/Os, there are numerous run-time issues that make these estimates extemely rough. For example, caching of data in the buffer pool can radically alter the profile of physical data accesses.

- While a database administrator may have access to certain kinds of run-time performance data about queries, such data is relatively difficult to gather systematically without support from a database utility. Further, performance data typically does not associate I/O behavior with data objects within a query. Thus it is generally not possible to apportion the performance numbers of a query to the individual objects (tables, indices, materialized views, etc.) used by the query.

Our goal in this paper is to overcome these difficulties so that a database administrator like Bob can make informed decisions about object placement, and make good use of limited SSD resources.

We have designed and implemented a tool that gathers run-time statistics on I/O behavior from an operational database residing on hard disk(s). By creating a benchmark installation in which each object is in its own tablespace, we are able to gather information about the random I/O behavior of each object without modifying the database engine code. We profile the application workload in this benchmark installation to gather statistics for each database object.

We calibrate each SSD device off-line to determine its sequential and random I/O performance for reads and (separately) for writes.

Based on the workload statistics and the device characteristics, we determine which objects would get the "best bang for the buck" by ranking them according to the expected performance improvement divided by the size of the object. We use a greedy heuristic to choose the highest-ranked objects one by one (in rank order) until no more objects fit within our SSD space budget. These chosen objects are our candidates for placement on the SSD device. We can create a variety of configurations with different space-time tradeoffs by varying the SSD budget. We also compare the greedy heuristic with a dynamic programming approach, which can generate better configurations for a larger variety of SSD space budgets at the expense of higher resource consumption (memory and CPU).

To the best of our knowledge, ours is the first work that shows how to cost-effectively make efficient use of available SSD resources in an "industrial strength" database system, and to provide guidance to database administrators about the benefits to be expected from the purchase of SSD devices.

We evaluate the performance of the placement advisor on TPC-C and TPC-H-like workloads. We construct the workloads to model a transactional database and an operational data store, respectively, that are expected to support both interactive queries and updates. The TPC-H queries are modified somewhat to query smaller fragments of the database, as might be more typical of an operational data store relative to a (read-mostly) data warehouse. When a relatively small amount of data is consulted for a query, random I/O performance becomes more important.[2]

In the evaluation, we use a database instance with access to both a hard disk and an SSD, and place the data according to one of several methods. We compare our method with three other options:

1. Place all data on the hard disk.

2. Place all indexes on the SSD, and all base tables on the hard disk.

3. Place all data on the SSD.

The second option is based on the observation that index accesses often involve random I/O. This option is likely to be the one chosen by a database administrator (DBA) in the absence of any design advisor tool.

Our results show that we can obtain significant speedups relative to the default strategy of placing all of the data on the hard disk. For example, on the TPC-H database, the hard-disk-based placement takes 139 minutes. We generate a set of configurations with various space-time trade-offs;

[2]Our technique is not limited to operational data stores, or to workloads dominated by random I/O performance. Because our SSD device and hard disk happened to have very similar sequential read bandwidth, our specific SSD provided little improvement for scan-dominated workloads. However, SSDs with faster sequential I/O are available, and would (at a higher cost) provide speedups for sequential access too.

one of these recommended configurations takes just 68 minutes using 1.8 GB of SSD space. These results are much better than the naive strategy of placing just the indexes on the SSD, which for the TPC-H database takes 117 minutes, and uses 5.5 GB of SSD space. Placing all data on the SSD is fastest, taking 53 minutes. However, this configuration requires 39 GB of SSD space, an order of magnitude more than our recommended placement.

The various space-time trade-offs allow a DBA to make choices about how much to invest in SSDs. If time is very important, then an organization might spend more on SSD devices in order to obtain even moderate marginal gains. Nevertheless, even a wealthy organization should not spend money on additional SSDs when the marginal gains are close to zero.

It is common practice to use only a small fraction of a physical hard disk for database data (typically just the outer tracks) so that seek times are small, a technique known as "short-stroking" [23]. If additional space is needed, a performance-sensitive DBA might choose to buy an extra hard disk rather than use the remaining space on existing disks. That way, each disk still has shorter seek times, and multiple disk heads can be seeking in parallel.[3] In this context, using an SSD for randomly accessed data can have secondary cost benefits. By moving the randomly accessed data to the SSD, the hard disks are left with data that tends to be accessed sequentially. As a result, it might be possible to use more tracks of each disk without incurring substantial extra seek time, and fewer disks would be needed. Additionally, by reducing the load on the disks, one might reduce overall power consumption because SSDs generally consume less power than hard disks.

In modern databases, storage characteristics are specified while creating the tablespaces so as to help the query optimizer choose a better query execution plan [10]. For instance, a query could be executed with *RID list fetch plan* if the storage device has very small average latency whereas the same query could have been executed with a *table scan plan* on a device with a higher average latency. Since the placement advisor considers the estimates from the profiling phase where the objects are placed on the HDD, the total execution times may not exactly match the expected execution times after the objects are moved to the SSD. Due to the changes in the plans, the workload execution may take shorter or longer than what is expected.[4]

The placement advisor could also be used to estimate and compare the performance benefits if different SSD devices are being considered for purchase. There are many different SSD products on the market, with very different performance and cost characteristics. One major distinction between these products is the interface that is used to connect the devices to the servers. Some of these options are: Fibrechannel, parallel SCSI, Parallel ATA, Bus (includes PCI, PMC, SBus, VME etc), Serial Attached SCSI (SAS) [13].

Although it seems advantageous to have plenty of options, it may not be easy to determine which option is best suited for a specific system. This is because for each device, the cost to provide data throughput for random and sequential reads and writes will vary. Consider the following hypothetical scenario in which the Object Placement Advisor would be helpful in comparing different SSD products.

Suppose two alternative SSD devices with specifications given below are under consideration for improving the performance of a database management system. Product B provides 5 times faster sequential access compared to Product A, while the random access speed of Product A is twice that of Product B. As for the price per GB of storage, Product B is twice as expensive as Product A. Given these specifications, the Object Placement Advisor might yield the chart given in Figure 1 after collecting profiling results of a workload running in the DBMS. If the budget is less than $60,000, then purchasing Product A would be a cost effective decision since it provides more improvement per dollar relative to B. If we have an SSD budget of $100,000 or more, then Product B would be preferred since the total performance gain of purchasing more of Product A would be less. With the help of this chart, one can determine which product minimizes cost and thus provides a higher utility.

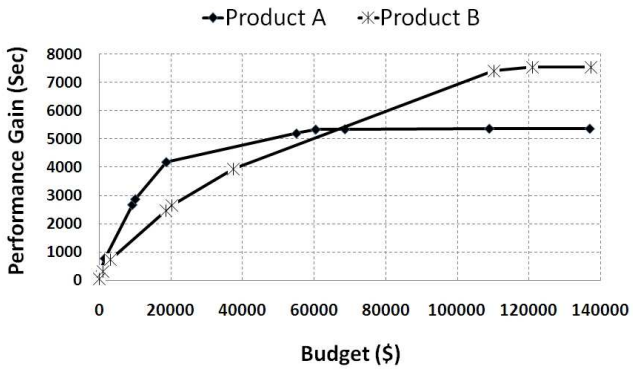

Figure 1: Performance comparison of different SSD products

The remainder of the paper is organized as follows: first, we introduce the Object Placement Advisor in detail; then, in Section 3, we present the results of the experiments conducted on the IBM DB2 DBMS with queries and schemas based on the TPC-H and TPC-C benchmarks; we present an overview of the related work in Section 4; finally, we conclude in Section 5.

2. OBJECT PLACEMENT ADVISOR

Similar to "REORG", a built-in utility in DB2 [10], Object Placement Advisor is proposed as an off-line tool. The procedure of obtaining an optimal placement strategy consists of two phases: a "Profiling phase" and a "Decision phase". Based on the collected run time statistics in the profiling phase, the estimated performance gain from moving each object from the HDD to the SSD is computed. Later on, these estimates are used in the decision phase to suggest an object placement plan.

The proposed database environment is illustrated in Figure 2. Suppose that we have a database management system processing the incoming queries of a workload. The database includes hundreds of tables, materialized views and indexes

[3]If the DBA does this, then the cost of an SSD might actually be competitive per "useful" gigabyte with the cost of a hard disk.

[4]With a perfect optimizer, and an SSD having superior performance to a hard disk, one would only see an improvement as one moves to an SSD.

created on multiple tablespaces. Initially, these tablespaces are created on HDDs. On top of this storage system there is a database engine processing the queries and transactions coming from user applications. A monitoring tool attached to the database engine measures the time spent for both reading the pages from the storage device to the buffer pool and writing the dirty pages back to the disk. These measurements are sent to the placement advisor for evaluation. Having this profiling input, the Object Placement Advisor (OPA) outputs a Cost-Benefit graph showing different placement strategies and the corresponding performance gains for different capacity constraints. Based on these options, the database administrator determines the amount of SSD space to be purchased and moves the objects from the HDD to the SSD according to the suggested placement plan.

In the following sections we will first discuss how profiling data is collected while running a query workload and then describe how to generate alternative placement strategies based on the estimated improvements.

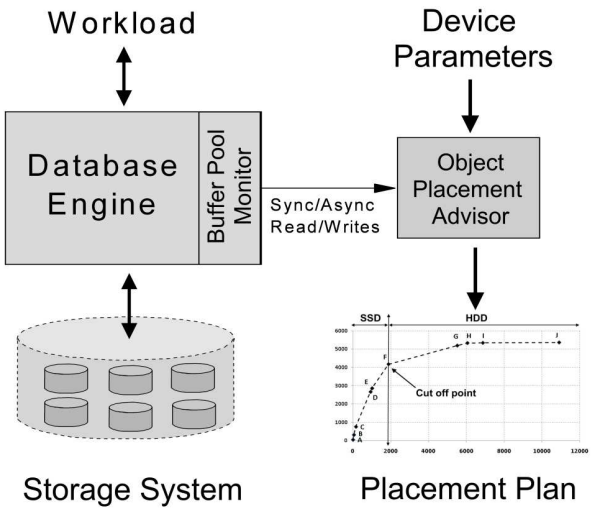

Figure 2: Object Placement Advisor (OPA) illustration

2.1 Profiling Phase

The goal of the profiling phase is to collect profiling information about the disk access costs for each database object during the execution of a query workload. This information includes the number of physical disk accesses such as sequential and random reads and writes and average page access costs for all objects in a single tablespace.

IBM DB2's snapshot utility is used to collect the profiling data at the table space level [10]. Using this utility, one can monitor activities on all buffer pools, tablespaces, and locks etc. Some of the parameters included in a typical output of a buffer pool snapshot are given in Table 1.

These parameters are used to measure the average sequential and random disk access costs of the pages pertaining to the objects as described below.

2.1.1 Average sequential access cost of a page

The parameters obtained from the snapshot report can be used to find the cost of sequential accesses. Whenever

Table 1: A sample buffer pool snapshot report for a tablespace.

Bufferpool name = IBMDEFAULTBP
Database name = TPCH
Snapshot timestamp = 07/17/2008 12:19:14.265625
Buffer pool data logical reads = 98
Buffer pool data physical reads = 27
Buffer pool temporary data logical reads = 0
Buffer pool temporary data physical reads = 0
Buffer pool data writes = 2
Buffer pool index logical reads = 214
Buffer pool index physical reads = 91
Buffer pool temporary index logical reads = 0
Buffer pool temporary index physical reads = 0
Total buffer pool read time (ms) = 947
Total buffer pool write time (ms) = 3
Asynchronous pool data page reads = 0
Asynchronous pool data page writes = 0
Buffer pool index writes = 0
Asynchronous pool index page reads = 0
Asynchronous pool index page writes = 0
Total elapsed asynchronous read time = 0
Total elapsed asynchronous write time = 0
Asynchronous data read requests = 0
Asynchronous index read requests = 0

the pages of an object need to be read sequentially from the disk, DB2 uses prefetchers which issue asynchronous read requests.

Agents of the application send these asynchronous requests to a common prefetch queue. As prefetchers become available, they fulfill these requests to fetch the requested pages from the disk into the buffer pool [10]. Therefore, asynchronous access parameters can be used to find the sequential read cost for the i^{th} table space. Similarly, the sequential write cost is attributed to the asynchronous write cost.

The *Total elapsed asynchronous read time* in Table 1 represents the total time spent for sequential read operations. The *Asynchronous pool data page reads* value, on the other hand, provides the number of pages of a data object read sequentially. Dividing the first parameter to the second yields the *Average sequential access cost of a page* for a data object. For index objects the same method is applied except that the *Asynchronous pool data page reads* is replaced with the *Asynchronous pool index page reads*. For write operations, a similar methodology is applied for both data and index objects.

2.1.2 Average random access cost of a page

DB2 does synchronous I/O requests wherever it is not able to make the requests asynchronously (e.g., the range of the pages to be retrieved is relatively small). Therefore, synchronous access costs can be used to measure the total random access cost for both data and index objects.

The buffer pool snapshot report does not include the synchronous I/O costs and the number of synchronous page accesses. However, these parameters can be computed using other parameters listed in Table 1. The total random read cost of the pages of an object is computed by subtracting the *Total elapsed asynchronous read time* from the *Total buffer pool read time*. This is applicable to the index objects as well as the data objects.

By subtracting the *Asynchronous pool data page reads* from the *Buffer pool data physical reads*, we obtain the *Synchronous pool data page reads* which is the number of pages read randomly from the disk for the data objects. To compute the same parameter for the index objects, the *Asyn-*

chronous pool index page reads is subtracted from the *Buffer pool index physical reads*.

To obtain the *Average random access cost of a page* for a data object, the *Total elapsed synchronous read time* is divided by the *Synchronous pool data page reads*. The same procedure is repeated to compute the *Average random access cost of a page* for an index object. For all write related parameters, the same procedure is applied by replacing the read related parameters with write related parameters.

2.1.3 SSD Device Characteristics

Before making placement decisions, the advisor needs to know the performance characteristics of the candidate SSD device(s). We measure the read and write performance of each device using a small set of straightforward sequential and random I/O microbenchmarks. In these microbenchmarks, we use a page size equal to DB2's page size. The sequential and random per-page costs will be used during the decision phase.

2.2 Decision Phase

Once the profiling phase is completed, the profiling results including the number of physical page accesses and the average page access costs are forwarded to the OPA to be used in the decision phase. By considering the profiling data, the estimated improvements (gains) for each object are computed. Using these estimations and the storage costs of the objects, the decision problem is first represented as an instance of 0-1 knapsack problem. Then, one of the well known heuristics, the *dynamic programming technique* or the *greedy technique*, is applied to obtain alternative placement strategies and the corresponding performance gains for different capacity (or budget) constraints. A detailed comparison of various aspects of these techniques is provided in Section 2.2.3.

2.2.1 Computation of Gain For Each Object

Since we have placed each object in its own tablepsace, the statistics for that tablespace reflect the access pattern for that object alone. There are four types of measurements: sequential read (SR), sequential write (SW), random read (RR) and random write (RW). Let j be one of $\{SR, SW, RR, RW\}$. Then $n_j(i)$ denotes the number of observed events of type j, $h_j(i)$ denotes the observed time taken by the hard disk to perform a page of I/O of type j for object O_i, and s_j denotes the time taken by the SSD to perform a page of I/O of type j. Then the *gain* γ_i that can be obtained by moving the database object O_i from the hard disk to the SSD is given by

$$\gamma_i = \sum_{j \in \{SR,SW,RR,RW\}} n_j(i) \times (h_j(i) - s_j)$$

2.2.2 Provisioning Problem

Consider a database with n objects $\varphi = \{O_1, O_2, ..., O_n\}$ all stored on an HDD initially. For a given workload, relocating object O_i from the HDD to an SSD provides γ_i units of estimated improvement in terms of disk access time. The storage cost of the i^{th} object is c_i. The objective is to maximize the total improvement by moving certain objects to the SSD under the constraint of C units of SSD space. This is an instance of the classical 0-1 knapsack problem [6].

Dynamic Programming solution

Let $\kappa_{n,C} = \{O_1, O_2, ..., O_n : C\}$ denote the 0-1 knapsack problem. Let a subset S of objects be optimal for $\kappa_{n,C}$ and O_i be the highest numbered object in S.

Then $S^{'} = S - \{O_i\}$ is an optimal solution for subproblem $\kappa_{i-1,C-c_i} = \{O_1, O_2, ..., O_i : C - c_i\}$ with $v(S) = \gamma_i + v(S^{'})$ where $v(*)$ is the value of an optimal placement "$*$" (i.e. the total improvement that can be obtained using that placement plan).

Then a recursive definition for the value of optimal solution would be as follows:

Define $v\{i, c\}$ as the value of an optimal solution for $\kappa_{i,c} = \{O_1, O_2, ..., O_i : c\}$.

$$v[i,c] = \begin{cases} 0, & \text{if i = 0 or c = 0} \\ v[i-1,c], & \text{if } c_i \geq c \\ max\{\gamma_i + v[i-1, c-c_i], & \\ v[i-1,c]\}, & \text{Otherwise} \end{cases} \quad (1)$$

This recursive definition specifies that an optimal solution $S_{i,C}$ for $\kappa_{i,C}$ either contains O_i (i.e. $v[i,c] = \gamma_i + v[i-1, c-c_i]$) or does not contain O_i (i.e. $v[i,c] = v[i-1,c]$). If the object O_i is picked to be moved to the SSD, the gain will be γ_i and then another object can be picked from $\{O_1, O_2, ..., O_{i-1}\}$ up to the storage limit $c - c_i$ to get the improvement $v[i-1, c-c_i]$. If the object O_i is not picked, an object from $\{O_1, O_2, ..., O_{i-1}\}$ up to the weight limit c, can be picked to get the improvement $v[i-1, c]$. The better of these two choices should be made.

Based on the recursive definition described above, the dynamic programming solution can be implemented.

Greedy solution

The greedy solution requires computing a priority value for each object. The priority value of the i^{th} object is computed as[5]:

$$P_i = \frac{\gamma_i}{c_i} \quad (2)$$

After this computation, the objects are sorted by their priority values in descending order. In this sorted list, the i^{th} object is the i^{th} best candidate that is recommended to be placed on the SSD. If c_1 units of SSD space are planned to be purchased, the first object in the sorted list would be the best object to be moved to the SSD. If $c_1 + c_2$ units of SSD space are purchased, the first and second objects in the list would be the most recommended objects to be moved to the SSD. Similarly, the i^{th} object in the list should be placed on the SSD provided that there is enough space to move the objects with lower indices to the SSD.

Considering the budget(the amount of SSD space) and estimated total improvement, a cut-off point in the list is determined. All of the objects falling above this cut-off point are moved to the SSD while the rest of the objects are kept on the HDD.

Thus, because it needs to sort the objects by their priority values the greedy algorithm runs in $O(n \log n)$ time.

Using this strategy, the database administrator can do a budget planning and determine how much space should be purchased. The details of this technique will be discussed in Section 3.3.

[5]The priority value is the gain per unit of storage (i.e. seconds of saved I/O time per megabyte of SSD storage).

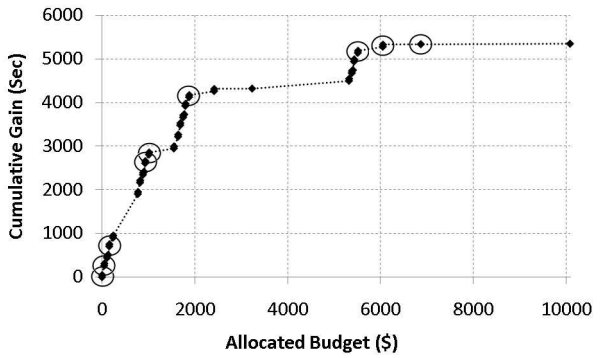

Figure 3: Cost - Benefit analysis using Dynamic Programming

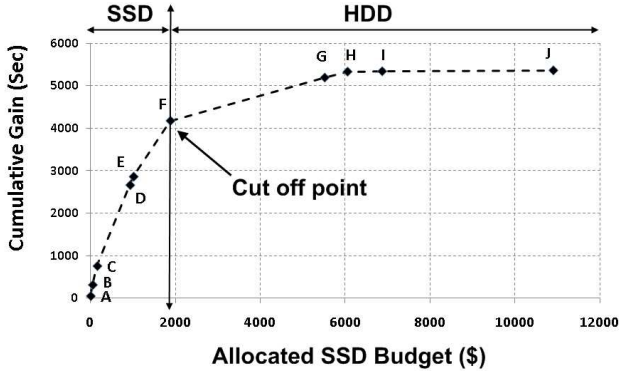

Figure 4: Cost - Benefit analysis using Greedy Technique

2.2.3 *Dynamic Programming vs. Greedy*

The Object Placement Advisor can use either a *dynamic programming technique* (DP) or a *greedy technique* in the decision phase. The dynamic programming technique provides the DBA with more placement options and the corresponding performance gains for different capacity constraints. On the other hand, the greedy technique finds the placement strategies that maximize the per unit gain (e.g. second/dollar, second/MB). The trade-off between the two techniques is that the greedy technique is polynomial-time bounded in the number of objects while the execution time of the DP technique is proportional to the knapsack size. Also, the DP technique requires an amount of main memory proportional to the knapsack size multiplied by the number of objects, while the greedy technique's memory requirements are proportional only to the number of objects.

In Figure 3 and Figure 4, two cost-benefit graphs corresponding to the DP and Greedy Techniques respectively are given. These results were obtained by simulating an Operational Data Store environment described in Section 3.3. Each point in these graphs represents different object placement strategies. The X-axis represents the total cost of employing a particular placement strategy while the Y-axis represents the total improvement in the workload execution time. Once a placement plan is chosen, the suggested objects are moved to the SSD while the rest of the objects are kept on the HDD. Note that in Figure 4, the total gain is a non-decreasing concave function of the total budget. This is because in the greedy technique the objects are first sorted by their unit gains and then at each point less valuable objects are added to the placement plan. The DP solution, on the other hand, includes not only the placement plans suggested by the greedy technique but also the other plans that provide more overall utility but less per unit gain. The circled points in Figure 3 corresponds to the placement plans suggested by the greedy technique.

3. EXPERIMENTS

To evaluate the effectiveness of the proposed technique, several experiments are conducted using TPC-H and TPC-C benchmarks on the IBM DB2 database server. Before discussing the main experiment results, we will first describe the hardware and software specifications and present results from a preliminary experiment.

3.1 Hardware & Software Specifications

The system that is used to run all the experiments has an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz with 8MB L2 cache and 4GB memory. Hardware specifications are given in Table 2.

Table 2: Hardware specifications

	HDD	SSD
Brand:	Seagate	Samsung
Storage capacity:	1 TB	64 GB
Interface:	SATA	SATA
RPM:	7200	N/A
Cache size:	32 MB	64 MB
Sequential Access - Read:	105MB/s	100MB/s
Sequential Access - Write:	30MB/s	80MB/s
Average latency - Read:	<8.5 msec	0.2 msec
Average latency - Write:	<9.5 msec	0.4 msec

All experiments are conducted on a 64 bit Fedora 8 (Linux kernel 2.6.24) operating system. IBM DB2 V.9 is used as the DBMS software. To monitor the random access cost of the database objects, each object is created on its own tablespace.

3.2 Impact of disk access behavior on response time

To observe the impact of mechanical movements in HDDs on data retrieval cost we implemented a simple application that reads 160MB of data in total from a 6 GB file. Given an input argument δ, the application reads the 160MB of data with jumps between two consecutive accesses where the jump size is computed as δ x 16KB. For instance if $\delta = 3$, the application reads 16KB data chunks from the offsets ρ, $\rho + (3 * 16KB)$, $\rho + (6 * 16KB)$, $\rho + (9 * 16KB)$ .. of the file where ρ is the file offset of the first read page.

At each run, we specified increasing jump sizes and measured the execution times on both the HDD and SSD. As the results in Figure 5 indicate, the page retrieval cost on the HDD depends on the distance between each page request while the retrieval cost is independent of the jump size on the SSD since there is no mechanical movement.

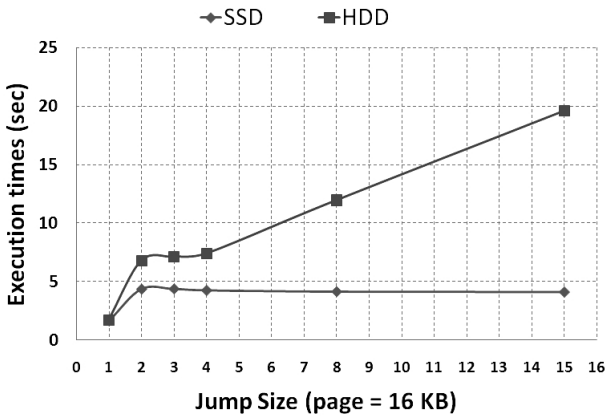

Figure 5: Reading 160 MB from a 6 GB file sequentially with jumps

This is essential for the object placement advisor, because it enables us to use a constant for the random access time on the SSD in the computation of estimated gains (see Section 2.2.1). Note that while the average page access times for each object stored on the HDD are available from the profiling phase, their SSD counterparts are not. Had the SSD random page access time not been constant, we would not have been able to acurately estimate the gains obtained by relocating each object to the SSD.

3.3 TPC-H based Experiments

The TPC-H benchmark is a decision support benchmark widely used in the database community to assess the performance of very large database management systems [31]. The benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and provide answers to critical business questions. Using the TPC-H schema and queries, we prepared an Operational Data Store (ODS) environment [9, 11, 26] to demonstrate the effectiveness of the Object Placement Advisor.

Preparation of the experiment bed and workload: In the TPC-H experiments each database is created using the TPC-H schema depicted in Figure 6. The database is comprised of 16 objects including 8 data objects and 8 index objects of the 8 relations. While generating the data, a scale factor of 30 is used. In total, 39GB of disk space are used to create the TPC-H databases.

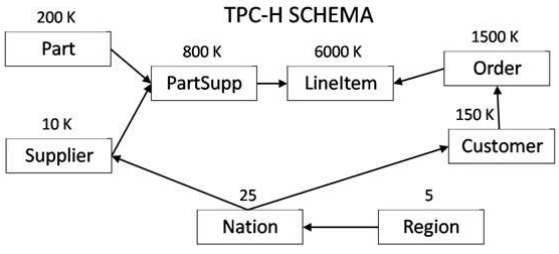

Figure 6: TPC-H Schema

The workload that is used in the experiments is constructed starting from 5 TPC-H queries (Query # 2, 5, 9, 11, 17) with the objective of maximizing the number of objects accessed during the execution of each query. We subsequently modified these queries to simulate an Operational Data Store (ODS) environment where some of the queries in the workload require processing large ranges of data while others process smaller ranges.

The major difference between an ODS and a data warehouse (DW) is that the former is used for short-term, mission-critical decisions while the latter is used for medium and long-range decisions. The data in a DW typically spans a five to ten years horizon while an ODS contains data that covers a range of 60 to 90 days or even shorter [9, 11]. In order to simulate an ODS environment, more predicates are added to the "where" clause of the TPC-H queries. This in turn, reduces the number of rows returned. As a result, we obtained a workload comprising of random and sequential accesses. The following query provides an example for this modification:

```
select n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey
and l_suppkey = s_suppkey and c_nationkey = s_nationkey
and s_nationkey = n_nationkey and n_regionkey = r_regionkey
and r_name = 'america' and o_orderdate ≥ '1993-01-01'
and o_orderdate ≤ '2000-01-01'
and o_orderkey between 170261658 and 170271658
group by n_name order by revenue desc
```

In this query, the predicate "o_orderkey between 170261658 AND 170271658" is added to the original query to reduce the range of the data that has been accessed. In order to reduce the bufferpool hit ratio, the predicate values are randomly drawn from a uniform distribution with range (0, n) where n is the maximum in the domain of the predicate attribute. Using this strategy, a workload of 1000 queries is prepared.

Table 3: Profiling Phase (Computation of Estimated Gains)

Object Name	Bufferpool Access Time (msec)	Estimated Gain (msec)	Space Requirement (MB)	Priority Value
PART_DAT	1492096	1314731	865.703	1518.686
SUPPLIER_DAT	304233	263924	46.563	5668.171
PARTSUPP_DAT	4091521	1015276	3632.719	279.481
CUSTOMER_DAT	2039413	1903452	772.344	2464.514
ORDERS_DAT	16716	9648	5071.141	1.903
LINEITEM_DAT	97456	85804	23758.59	3.611
PART_IX	493267	442893	108.766	4071.993
SUPPLIER_IX	51225	47810	5.453	8767.450
PARTSUPP_IX	212758	198517	81.578	2433.461
CUSTOMER_IX	149978	139979	537.250	260.548
ORDERS_IX	9695	9049	815.703	11.093
LINEITEM_IX	22443	20947	4029.344	5.199

In the profiling phase, the database is created on the HDD. As described in section 2.1, for each object, a separate tablespace is created so as to monitor disk access times. The monitoring results are given in Table 3 [6]. The object names are stated in the first column. "Dat" and "IX" represent

[6]The objects Nation_IX, Nation_DAT, Region_IX and Region_DAT are omitted from these tables since the storage requirements of these objects are fairly small.

the data part and the index part of the relations respectively. The total buffer pool disk access costs of the objects are given in the second column. Using the bufferpool snapshot reports, the estimated gains are computed (given in the third column of the table). As discussed earlier "gain of the i^{th} object" is the estimated improvement in terms of the response time when the i^{th} object is moved from the HDD to the SSD. Using the gain and space requirement given in the third and fourth columns of Table 3 respectively, the priority values are computed (fifth column of Table 3).

In the decision phase, we use the greedy knapsack technique to determine the size of the SSD that should be purchased and the objects to be moved to the SSD. To employ this technique, the objects in Table 4 are first sorted by their priority values in descending order. The second and third columns in the table represent the cumulative storage cost and estimated cumulative gain of the objects respectively. Using this table, an optimal placement decision can be made as follows: if the database administrator (DBA) decides to purchase 5.5 MB of SSD space, the optimal placement strategy would require moving SUPPLIER_IX to the SSD since it has the highest priority value in the list. In this case, the total improvement in terms of the workload execution time would be 47,810 milliseconds. If the DBA decides to increase its budget (knapsack size) and to purchase 52 MB of SSD space, the best placement strategy would be to move the SUPPLIER_IX as well as SUPPLIER_DAT objects to the SSD. Moving these objects would improve the execution time by 311,734 milliseconds. Similarly, if 161 MB of SSD space are purchased, the third object to be included in the knapsack would be PART_IX which would provide an improvement of 754,627 milliseconds.

Table 4: Decision Phase Results (The objects are sorted by priority values)

Option	Object Name	Cumulative Space (MB)	Estimated Cumulative Gain (msec)	Priority Value
A	SUPPLIER_IX	5.5	47810	8767.45
B	SUPPLIER_DAT	52.0	311734	5668.11
C	PART_IX	160.8	754627	4071.99
D	CUSTOMER_DAT	933.1	2658079	2464.54
E	PARTSUPP_IX	1014.7	2856596	2433.41
F	PART_DAT	1880.4	4171328	1518.68
G	PARTSUPP_DAT	5513.1	5186603	279.48
H	CUSTOMER_IX	6050.4	5326583	260.54
I	ORDERS_IX	6866.1	5335631	11.09
J	LINEITEM_IX	10895.4	5356578	5.19
K	LINTEITEM_DAT	34654.0	5442382	3.61
L	ORDERS_DAT	39725.2	5452030	1.90

In Figure 4 the estimated cumulative gain (E.C.G.) is plotted against the budget allocated to buying SSD space (the second and the third columns of Table 4). For simplicity, we assume that 1MB of SSD space costs $1 (in reality it is significantly lower, currently as low as $0.01 for 1MB). The figure suggests that E.C.G. is an increasing and concave function of the allocated budget implying that objects with the highest priorities are selected first to be moved to the knapsack and as the knapsack size increases, less valuable objects are picked.

The database administrator employs a cost benefit analysis in determining the SSD size. Note that the set of feasible storage options are the points on the horizontal axis in Figure 4 corresponding to points A,B,C.. etc.. This is because an object should either be located on the SSD or on the HDD. Consider a database administrator with a budget of $3000 or less. The SSD space options available to this administrator will be those on the horizontal axis corresponding to points A, B, C, D, E and F in Figure 4. Since the marginal gain from an additional dollar spent on SSD space is strictly positive at all available options to the left of $3000, the best decision for the DBA is to buy an SSD space of 1.8GB which is the horizontal axis component of point F. Therefore in this scenario, we moved the first 6 objects; Supplier_IX, Supplier_Dat, Part_IX, Customer_Dat, PartSupp_IX, Part_Dat to the SSD and left the rest of the objects on the HDD.

Note that if the DBA has a budget of $6000 or more, he would be better off by selecting H as the cut off point since there is not much marginal gain from spending an additional dollar after this point.

In addition to the database that is created for collecting profiling data, three more databases are created. In the second database the indexes are moved to the SSD while the data objects are created on the HDD, for the purposes of measuring the effectiveness of the naive approach. The third database is created using the object placement strategy suggested by the OPA. According to this plan, the first 6 objects in Table 4 are moved to the SSD while the remaining objects are kept on the HDD. In the fourth database, all objects are created on the SSD, to see the gain when all objects are moved to the SSD.

The workload execution times that we obtained in four separate experiment setups are shown in Figure 7. Applying the naive approach yields a 16% total improvement with a cost of 5.5GB SSD space. On the other hand, placing the objects considering the suggested priority list of the OPA improves the workload execution time by 51% with a cost of 1.8GB SSD space. The rightmost bar in Figure 7 corresponds to the workload execution time when all the objects are placed on the SSD. The minimum workload execution time is obtained in this setup but with a large SSD space of 39GB.

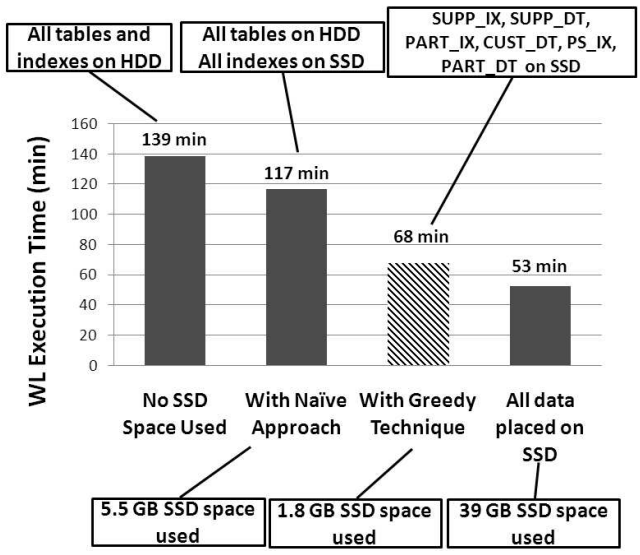

Figure 7: TPC-H Results

As observed in this experiment, the object placement advisor helps to make a cost effective decision that maximizes the benefit with a reasonable cost.

3.4 TPC-C Experiments

TPC-C is a popular benchmark for comparing on-line transaction processing (OLTP) performance on various hardware and software configurations. TPC-C simulates a complete computing environment where multiple users execute transactions against a database. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. The transactions do update, insert, delete, and abort operations and numerous primary and secondary key accesses [30].

The transactions operate against a database of nine tables. The schema is shown in Figure 8. The numbers in the entity blocks represent the cardinality of the tables (number of rows). These numbers are factored by W, the number of Warehouses, to illustrate the database scaling.

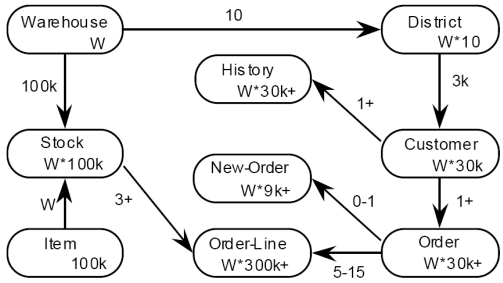

Figure 8: TPC-C Schema

Majority of the objects in the TPC-C database are accessed randomly. We observed that 94 % of the disk access cost is caused by random accesses. Therefore, moving all of the objects to the SSD seems to be the best solution. However, depending on the frequency of the transactions in the workload, some objects might be accessed more frequently than others. In this experiment the object placement advisor, helps to identify these frequently accessed objects.

In our experiments, the scaling factor is set to 20 Warehouses during the data generation. With this scaling factor each database occupies a total of 2.2GB disk space. In the first experiment, all of the objects are created on the HDD. The profiling data is collected while executing concurrent transactions issued by 15 clients. As in the TPC-H experiments, we measured the gain of the objects and sorted the objects by their priority values. Based on these monitoring results, the objects New_Order_DT, Item_IX, Item_DT, Stock_IX, Customer_IX, and Stock_DT are moved to the SSD while the remaining objects are kept in the HDD.

The TPC-C benchmark results are compared with tpmC rate which is the average number of concurrent transactions completed in one minute. We used tpmC rate to examine the system performance when different object placement strategies are employed. The average tpmC rate that we obtained in four separate experiment setups are shown in Figure 9. The leftmost bar corresponds to the number of transactions executed when all the objects are placed on the HDD. The second bar corresponds to the naive approach in which the index objects are moved to the SSD and the data objects to the HDD. As can be seen, the tpmC rates in the third and fourth bars are significantly larger than the first two bars. The third bar corresponds to the number of transactions executed when the objects are moved to the SSD considering the priority list provided by the OPA, while the last bar represents the number of transactions when all of the objects are moved to the SSD.

The results indicate that even though the vast majority of the objects in a DBMS are accessed randomly, moving all of the objects to the SSD may not be the most cost effective decision. As it is seen in this scenario, a satisfactory improvement in the transaction rate can be achieved with one third of the total storage requirement of the objects. On the other hand, the idea of moving only the index objects to the SSD does not seem to be an effective solution as the tpmC rate is almost identical to the one where no SSD space is used.

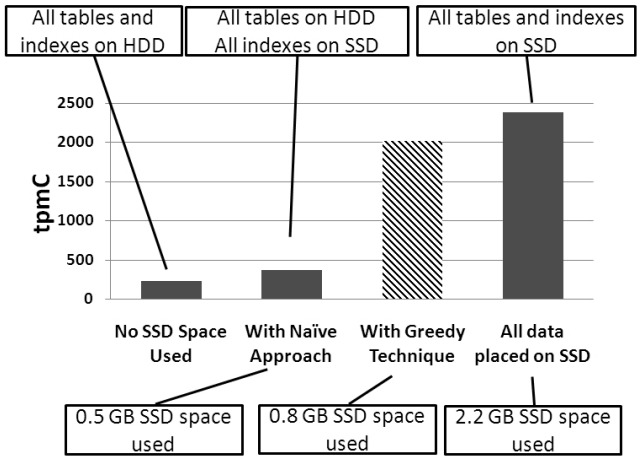

Figure 9: TPC-C Results - Y axis represents the number of transactions per minute with 15 clients

4. RELATED WORK

The data placement problem is one of the oldest problems in Computer Science. Indeed, as soon as a computer system has access to a multitude of storage devices of varying performance and cost, it becomes important to decide where to place each file, so as to obtain the best performance for the money. Of particular interest is data allocation in storage hierarchies, where different data sets need to be placed at different levels of a storage hierarchy and moved across levels whenever their access patterns change [19].

More recent work in this area is on data placement for large-scale distributed caches [15] that focuses on reducing access costs to remote caches at runtime. Other recent work investigates the physical layout of data on disks based on disk access characteristics [27] which is orthogonal to our problem and at a lower level. Placement of data tuples on different disks of a shared nothing RDBMS for the purpose of increased query parallelism ("declustering") is investigated in [21] among others. While they consider skew in the data and access distribution, they assume homogeneous

disks and can therefore not exploit the special properties of SSDs. A more theoretical treatment of the table and index placement problem for heterogeneous sets of disks is discussed by Aggarwal et al. [1]. They present algorithms to increase disk access parallelism while at the same time reducing the random access cost in a storage environment with different random/sequential access costs. The difference to our algorithm is that we measure actual table access patterns for given workloads which allows us to capture subtle interactions between concurrently running queries.

Modern commercial database systems have recommender tools designed to assist a database administrator in the physical database design phase. Examples for such tools are the SQL Access Advisor [7] in Oracle 10g that helps in the selection of indexes and materialized views, and the Database Tuning Advisor [2, 3, 5] in Microsoft SQL Server that proposes indexes, views, and horizontal partitioning. Another example is the IBM DB2 Design Advisor [33] that analyzes a given workload and recommends a set of materialized query tables (MQTs) and indexes to maximize the workload performance. None of these recommender tools provide advice on placing tables on storage devices with varying access characteristics.

The IBM DB2 Design Advisor's capability has been recently extended with a data placement advisor utility [18], which recommends an optimized placement of MQTs in a multi-tiered setting. This is similar in spirit with our approach, albeit with a different goal, namely reducing latencies by placing part of the data closer to the applications. The solution is based on analyzing the ratio between the anticipated benefit provided by each MQT versus its size. One distinction from our problem is the presence of dependencies between individual MQTs which our setting does not exhibit: the placement decision for each object does not depend on the placement of other objects.

With the advent of solid state storage, there has been increasing interest in the database research community for the exploitation of this new technology. Some approaches can be used with standard, off-the-shelf DBMS products while others require changes in the DBMS engine.

In the first category (where our work also belongs), the use of SSD's for temporary tablespace has been shown to improve performance by more than an order of magnitude [17]. While this is certainly a valuable observation, it is subsumed by our approach as the temporary tablespace can be included in the list of candidate objects and the Object Placement Advisor can decide whether or not to place it on SSD based on its calculated priority value.

In the second category, the use of the PAX column-major storage layout [4] has been exploited for a novel random-read efficient join algorithm, RARE-join [28]. Another kind of engine adaptation in the form of flash-sensitive B+-Trees have been proposed in [22, 25, 32]. Also, a novel in-page logging scheme has been proposed in [16] in order to circumvent the non in-place update limitations of flash. Clearly, since SSD characteristics are quite different from those of traditional hard drives, we share the view of these researchers that engine data structures and algorithms need to be re-examined, if one wants to take full advantage of solid state storage.

More closely related to our work, Koltsidas and Viglas [14] study how to adapt the storage layer for use with a combination of both flash and magnetic drives. They assume that individual pages can be placed either on the flash device or on the magnetic device. They show how one might change the design of buffer pool replacement algorithms so that pages are placed on the better-suited device for that page. Our work differs from this work in several respects. First, their study applies only to MLC flash devices. They argue that since MLC devices are likely to be cheaper than SLC devices, it is more likely that such devices will become the commodity device of the future. However, MLC devices are far less reliable than SLC devices [12, 20]. As a result, we expect that at least in the medium-term, SLC drives will be the device of choice for enterprise-level storage. Second, they assume that the flash disk is ten times faster than the magnetic disk for random reads, but ten times slower for random writes. These performance characteristics do not carry over to SLC devices. In particular, on SLC devices like the one we employed, both random reads and random writes are an order of magnitude faster on the flash device than on the magnetic disk. The MLC/SLC distinction mentioned above has critical implications for the optimization problem studied. For Koltsidis and Viglas, the aim is to spread pages between the two devices so that the read-intensive pages tend to be on the flash device, and the write-intensive pages tend to be on the disk device. If a flash device were to dominate a magnetic device in both read and write performance, their placement algorithms would put all of the data on the flash device; no capacity constraints are considered. In contrast, our metrics are primarily based on cost/capacity constraints, assuming that the flash device is more expensive per gigabyte than the magnetic device. We aim to find the part of the database that would benefit the most from a move to the flash device. Furthermore, Koltsidas and Viglas assume that storage decisions can be made at a page granularity, i.e., that the choice of whether to store a page on the flash device is independent of the choice for all other pages. Clearly this assumption requires that the storage manager be modified to implement such a placement method. Further, there are disadvantages to such a method that are not discussed by the authors. For example, a table that is initially stored contiguously on the magnetic device could end up being fragmented, drastically reducing the sequential I/O throughput. In contrast, our approach requires no changes to the storage manager. Whole database structures (tables, indexes) are placed on each device as appropriate. Our work can, today, be directly applied to workloads running on commercial database systems. Finally, Koltsidas and Viglas do not explicitly distinguish between sequential and random I/O in their models, even though sequential I/O is very common for database workloads. According to their analysis, a magnetic disk is superior to an MLC flash disk for both sequential reads and sequential writes. They state that one way to incorporate sequential I/O into the model would be to ask the database engine to supply hints to the buffer manager about the sequential nature of the access pattern. In contrast, our method explicitly accounts for the sequential/random I/O distinction when computing our gain function.

5. CONCLUSIONS AND FUTURE WORK

In this paper we described an object placement advisor, an out-of-engine tool which can be used to decide, for a given workload, which database objects should be placed on an SSD for best performance. The advisor uses snapshot data collected by DB2 to estimate the time savings that would be

realized by moving each object to an SSD. The placement problem is thus reduced to a knapsack problem, where the cost is the size of each object and the benefit is the estimated access time saved. The knapsack problem can subsequently be solved using any known heuristic, for example greedy or dynamic programming. By implementing the recommended placement configuration, we have observed significant performance improvements at a much lower storage cost when compared with a naive placement strategy.

While we have focused on SSD devices in this paper, our methods could equally well be applied to other new technologies that are developed for use as on-line persistent storage devices.

In future work, we plan to consider using more than two kinds of devices, such as having two different SSDs with different capabilities. The optimization problem we need to solve would be more involved. Having multiple types of devices may allow for specialized placement, matching the bottleneck I/O parameter (read or write; sequential or random) with the device that best handles such workloads.

For this paper, we only considered the performance characteristics when choosing a device over another. Other considerations may turn out to be just as important as raw performance in practice. For example, the mean-time to failure (MTTF) contributes directly to the total cost of ownership. Replacement is an indirect aspect of the cost of choosing a particular device. While early flash devices did wear out fairly quickly, recent SSD-grade devices use more reliable technology with wear-leveling that makes their expected lifetimes at least several years, even if used continuously. Energy consumption is yet another comparison factor.

Currently we do our initial measurements using the standard optimizer, which is not SSD-aware. We might be missing some opportunities because of the following kind of scenario: plan X is generated on the HD configuration, where X does a sequential scan; plan Y, which does random I/O is marginally worse on the HD configuration, and so is not generated. On the SSD system, the cost of plan X might be no better than the cost of plan X on the HD system. However, the cost of plan Y might be much better, and justify moving the corresponding data objects to the SSD. To overcome these limitations, the optimizer needs to be made aware of the specific characteristics of the SSD device. We plan to investigate this direction as part of future work.

Finally, while we only experimented with DB2 for this paper, we believe our solution could be used together with other DB systems. For example, Oracle's Automated Workload Repository [24] and SQL Server's Performace Studio [29] are both providing similar runtime statistics which can, in principle, be exploited by our Object Placement Utility.

6. ACKNOWLEDGMENTS

We would like to thank Murat Kantarcioglu for helpful discussions. We also thank the anonymous reviewers for their feedback.

7. REFERENCES

[1] G. Aggarwal, T. Feder, R. Motwani, R. Panigrahy, and A. Zhu. Algorithms for the database layout problem. In *ICDT*, pages 189–203, 2005.

[2] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005: demo. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 930–932, 2005.

[3] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 683–694, 2006.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance (best paper award). In *Proceedings of the 27th International Conference on Very Large Data Bases(VLDB '01)*, pages 169–180, Sept. 2001.

[5] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling. Self-tuning technology in microsoft sql server. *IEEE Data Eng. Bull.*, 22(2):20–26, 1999.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[7] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL tuning in Oracle 10g. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1098–1109, 2004.

[8] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*, page 6, 2007.

[9] P. Gray and H. J. Watson. Present and future directions in data warehousing. *SIGMIS Database*, 29(3):83–90, 1998.

[10] IBM DB2 Database for Linux, UNIX, and Windows Information Center. `http://publib.boulder.ibm.com`.

[11] W. H. Inmon. *Building the Operational Data Store*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[12] Z. Kerekes. Are mlc ssds ever safe in enterprise apps? `http://www.storagesearch.com/ssd-slc-mlc-notes.html`.

[13] Z. Kerekes. The ssd buyer preferences market report. `http://www.storagesearch.com/ssdsurvey.html`.

[14] I. Koltsidas and S. Viglas. Flashing up the storage layer. *PVLDB*, 1(1):514–525, 2008.

[15] M. R. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Trans. on Knowl. and Data Eng.*, 14(6):1317–1329, 2002.

[16] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, 2007.

[17] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, 2008.

[18] W.-S. Li, D. C. Zilio, V. S. Batra, C. Zuzarte, and I. Narang. Load balancing and data placement for multi-tiered database systems. *Data Knowl. Eng.*, 62(3):523–546, 2007.

[19] V. Y. Lum, M. E. Senko, C. P. Wang, and H. Ling. A cost oriented algorithm for data set allocation in storage hierarchies. *Communications of the ACM*, 18(6), June 1975.

[20] L. Mearian. Solid-state disk lackluster for laptops, pcs. `http://www.computerworld.com/action/article.do?command=printArticleBasic&articleId=9112065`.

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

[22] S. Nath and A. Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, 2007.

[23] J. A. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your place in the file system with gray-box techniques. In *In Proceedings of the USENIX Annual Technical Conference (USENIX 03*, pages 311–324. USENIX Association, 2003.

[24] Oracle Automated Workload Repository. `http://www.oracle.com/technology/deploy/performance/index.html`.

[25] K. A. Ross. Modeling the performance of algorithms on flash memory devices. In *DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 11–16, 2008.

[26] J. Samos, F. Saltor, J. Sistac, and A. Bardés. Database architecture for data warehousing: An evolutionary approach. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 746–756, 1998.

[27] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 22, 2002.

[28] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 17–24, 2008.

[29] Microsoft SQLServer Performance Studio. `http://www.microsoft.com/sqlserver/2008/en/us/wp-sql-2008-performance-scale.aspx`.

[30] TPC-C, On-Line Transaction Processing Benchmark. `http://www.tpc.org/tpcc/`.

[31] TPC-H, Decision Support Benchmark. `http://www.tpc.org/tpch/`.

[32] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.

[33] D. C. Zilio, C. Zuzarte, G. M. Lohman, H. Pirahesh, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with ibm db2 design advisor. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 180–188, 2004.