Automating Layout of Relational Databases

Sanjay Agrawal, Microsoft Research
Surajit Chaudhuri, Microsoft Research
Abhinandan Das, Cornell University
Vivek Narasayya, Microsoft Research

Abstract

The choice of database layout, i.e., how database objects such as tables and indexes are assigned to disk drives can significantly impact the I/O performance of the system. Today, DBAs typically rely on fully striping objects across all available disk drives as the basic mechanism for optimizing I/O performance. While full striping maximizes I/O parallelism, when query execution involves co-access of two or more large objects, e.g., a merge join of two tables, the above strategy may be sub-optimal due to the increased number of random I/O accesses on each disk drive. In this paper, we propose a framework for automating the choice of database layout for a given database that also takes into account the effects of co-accessed objects in the workload faced by the system. We formulate the above as an optimization problem and present an efficient solution to the problem that judiciously takes into account the trade-off between I/O parallelism and random I/O accesses. Our experiments on Microsoft SQL Server show the superior I/O performance of our techniques compared to the traditional approach of fully striping each database object across all disk drives.

1. Introduction

In today's enterprises, relational database systems (RDBMSs) play a crucial role in the back-end for storing and retrieving information. As databases continue to get larger, achieving good overall performance for queries and updates that execute against a database requires good I/O performance. The appropriate choice of access methods such as indexes and materialized views is an integral part of ensuring good I/O performance of queries that execute against the RDBMS. However, another significant factor affecting I/O performance of queries is *database layout*, i.e., how database *objects* such as tables, indexes, materialized views etc., are assigned to the available disk drives in the system.

Traditionally, enterprise databases have relied on solutions that spread out each database object uniformly over all available disk drives, thereby obtaining good I/O parallelism. A typical solution is to use one or more disk drives, each of which may itself be an array of disks (e.g., a RAID (Redundant Arrays of Inexpensive Disks) array), and then use *full striping* to spread each database object across all disk drives. Such a solution has the advantage that it is relatively easy to manage since the database administrator (DBA) does not have to be concerned about which disk drive(s) each object should be placed on. However, as the following example shows, for queries in which multiple large objects (tables or indexes) are accessed together during execution (e.g., queries in DSS applications), a solution that spreads each object over all available disk drives may perform sub-optimally.

Example 1. Consider queries Q_3 and Q_{10} of the TPC-H benchmark [15]. The execution plan of both these queries accesses the tables *lineitem* and *orders* together and performs a Merge Join. We measured the execution time of these queries on a 1GB TPC-H database on Microsoft SQL Server 2000 for the following two database layouts over a set of 8 disk drives: (1) Full striping: Each table was spread uniformly across every disk drive (2) *lineitem* was spread uniformly on 5 disk drives, *orders* was spread uniformly on the 3 *other* disk drives. Q_3 executed about 44% faster on the database layout (2) as compared to (1), and Q_{10} similarly executed about 36% faster. In both queries, the key factor that made the second database layout faster was that the objects that were co-accessed during the execution of each query (*lineitem* and *orders*) were on different disk drives, thereby eliminating a large number of random access I/Os that were incurred in the first database layout.♦

As illustrated by the above example, a database layout such as full striping, that is optimized for I/O parallelism may suffer in performance when the *workload* consists of queries and updates having significant co-access among objects. Thus, when determining a good database layout, there is a need to take into account the trade-off between benefit due to I/O parallelism and overhead due to random I/O accesses introduced by co-locating objects that are co-accessed during query execution. For workloads containing queries that co-access multiple objects, the gain in I/O performance by choosing an appropriate database layout other than full striping can be significant.

While the specific problem of high random I/O accesses due to large co-accessed objects could be reduced by modifying the query execution strategy inside the server (e.g., by issuing larger reads), in this paper we consider an alternative approach that allows us to also incorporate other aspects of database layout such as manageability and availability requirements, which are crucial for practical deployment of any solution. This paper makes the following contributions. We present a framework for specifying the *database layout problem* – i.e., the problem of automatically choosing a database

layout that is appropriate for the workload faced by a database system, while satisfying manageability and availability requirements. We develop a cost model for quantitatively capturing the above trade-off between I/O parallelism and random I/O accesses for a given workload. Such a cost model is essential to allow us to compare the relative "goodness" of two different database layouts for the workload. We show that the database layout problem can be formulated as an optimization problem, and establish that this problem is provably hard. We present a principled approach for solving the database layout problem that judiciously addresses the above based on characteristics of the workload. Finally, we demonstrate via experiments on Microsoft SQL Server 2000 that the database layouts chosen by our solution result in superior I/O performance than the solution of full striping (which only maximizes I/O parallelism).

This work was done in the context of the AutoAdmin project [1] at Microsoft Research. The goal of the AutoAdmin project is to reduce the total cost of owning a RDBMS by automating important and challenging database administrative tasks. The rest of this paper is structured as follows. In Section 2, we formulate the database layout problem as an optimization problem, and describe the architecture of our solution in Section 3. In Section 4, we show how we exploit information about the workload in our solution. Section 5 presents our model of the I/O performance of the workload, which is the metric that we wish to optimize. We describe the strategy for solving the optimization problem in Section 6, and in Section 7 we present results of experiments comparing our solution to the approach of full striping. We discuss related work in Section 8 and conclude in Section 9.

2. Problem Formulation

In this section, we present a framework for specifying the database layout problem. We first describe the two key concepts in our framework: (1) A database layout, and how it can be specified in today's commercial database systems. (2) Our model of the workload. We then present a formulation of the database layout problem and show how to include manageability and availability requirements into the formulation.

2.1 Database Layout

We assume that a relational database consists of a set of tables and physical design structures defined on the tables. The database *objects* that we consider include tables, indexes, materialized views, and in principle, other access methods that may be present in the database. We denote the set of *n* database objects in a database by $\{R_1, \ldots R_n\}$. The DBA is responsible for determining the placement of the database objects on the available set of *m* disk drives $\{D_1, \ldots D_m\}$. Each disk drive is a single addressable entity that itself could be comprised of a set of disks bound together into a disk array. For our purposes, the following properties of a disk drive D_j are relevant: *capacity* C_j (e.g., 8GB), average *seek time* Sj (e.g., 10msec), average *read transfer rate* TR_j (e.g., 10MB/sec) and average *write transfer rate* TW_j, and *availability property* $AVAIL_j$ which can take on one of the following values: {*None*, *Parity*, *Mirroring*}. For example, AVAIL property of a RAID 0 disk drive or a stand alone disk is *None*; AVAIL property of a RAID5 disk drive is *Parity*; and AVAIL property of a RAID 1 disk drive is *Mirroring*. We discuss the relevance of the availability property to the database layout problem in Section 2.3.

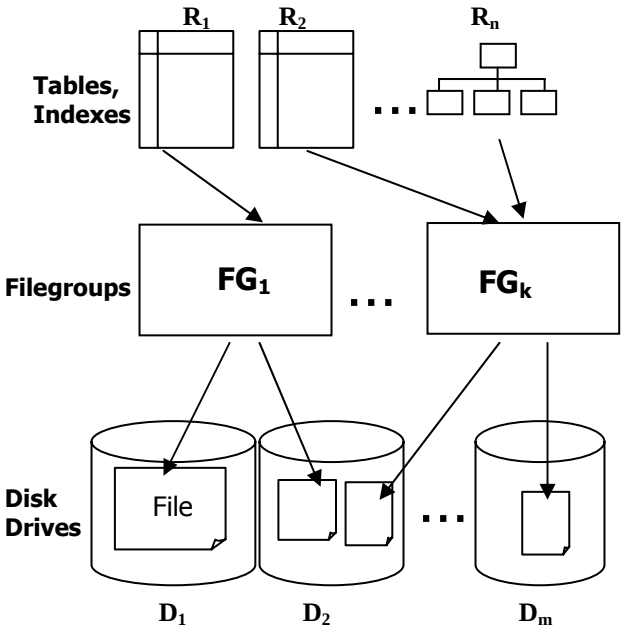

Figure 1: A Database Layout.

Today's commercial database systems allow the DBA the flexibility of allocating each object over multiple disk drives. For example, in Microsoft SQL Server 2000, an object can be allocated on multiple disk drives by defining a *filegroup*, and assigning the object to the filegroup as shown in Figure 1. A filegroup is a collection of files that are present on one or more disk drives (The concept of filegroups is similar to *tablespaces* in Oracle and IBM DB2). Each object can be assigned to exactly one filegroup, although it is possible to assign more than one object to a given filegroup. Finally, any set of filegroups may overlap in the set of disk drives over which they are defined. For example, in the figure, we note that disk drive D_2 is common to filegroups FG_1 and FG_k.

When an object is assigned to a filegroup that is defined over more than one disk drive, the storage engine component of the database system distributes the pages of

the object in a particular manner (e.g., round robin fashion) across the disk drives. The allocation is done not at the granularity of a page, but at the granularity of a *block*, (e.g., 8 pages in Microsoft SQL Server 2000). Furthermore, we can control the fraction of the total number of blocks of an object that is allocated to each disk drive. Thus, a database layout is an assignment of each database object to a filegroup, along with a specification of the fraction of the object that is allocated to each file in that filegroup. Since, for our purposes, each filegroup can itself be viewed as the set of disk drives on which it is defined, we equivalently define a database layout as follows:

Definition 1. **Database Layout**: A database layout is an assignment of each database object to a set of disk drives along with a specification of the fraction of the object that is allocated to each disk drive. ♦

Definition 2. **Valid Database Layout**: We define a database layout as *valid* if it satisfies the following two criteria: (1) For each disk, the database layout does not violate the capacity constraint of that disk. (2) Each object is allocated in its entirety.♦

Logically, a database layout is specified by a two-dimensional matrix where each row corresponds to an object and each column corresponds to a disk drive. The value of a cell x_{ij} ($0 \leq x_{ij} \leq 1$) in the matrix is the fraction of the total number of blocks of object R_i that is placed on disk drive D_j. We denote the size of object R_i by $|R_i|$ and the capacity in blocks of disk D_j by C_j. In terms of the above notation, a layout is valid if it satisfies the following three constraints (*n* is the number of objects and *m* is the number of disk drives).

$$\forall i \in [1 \ldots n], \forall j \in [1 \ldots m] \; x_{ij} \geq 0$$

$$\forall i \in [1 \ldots n] \sum_{j=1}^{m} x_{ij} = 1$$

$$\forall j \in [1 \ldots m] \sum_{i=1}^{n} |R_i| . x_{ij} \leq C_j$$

The first two constraints together ensure that each object is allocated sufficient disk space, and the third constraint ensures that the capacity constraint for each disk is not violated. Finally, we note that objects created temporarily during query execution can also have a significant impact on I/O performance (e.g., large sorts, hash joins). We can incorporate these effects by modeling temporary tables as objects in our formulation (which are stored in the *tempdb* database) with the constraint that all these objects should be stored on the same filegroup.

2.2 Workload

The appropriate choice of database layout depends on the nature of the workload faced by the system, i.e., the I/O access patterns of queries and updates that execute against the system. For example, for a single-table query that involves the scan of a large table (or index), it may be advantageous to define a layout in which the referenced table (or index) is allocated on a large number of disks. This is because the object can be scanned in *parallel* on all disks, thereby reducing the I/O response time for that query. On the other hand, if the query requires simultaneously accessing two or more large objects (e.g., a merge join of two tables), it may be better to allocate the objects on disjoint sets of disk drives. The reason is that if the two objects are co-located on a disk drive, then a potentially large number of random I/O accesses are introduced on that disk drive when the two objects are simultaneously accessed by the query, thereby making that disk a potential I/O bottleneck for the query.

In this paper, we assume that a workload is provided as input. We define a workload as a set of SQL DML statements, i.e., SELECT, INSERT, UPDATE and DELETE statements. Optionally, each statement Q in the workload may have associated with it a *weight* (denoted by w_Q) that signifies the importance of that statement in the workload. For example, weight may indicate the multiplicity of that statement in the workload. A representative workload for the system can be gathered using profiling tools available in modern commercial database systems, e.g., the SQL Server Profiler in Microsoft SQL Server. Alternatively, DBAs can specify a custom representative workload, e.g., an organization or industry specific benchmark. In Section 4 we show how given such a workload, we can extract the relevant access and co-access information about database objects.

Since we model the workload as a *set* of statements, we do not take into account the impact on database layout by statements that execute *concurrently* with one another. In particular, this has the effect of underestimating the amount of co-access between objects. Incorporating effects of concurrent query execution into the workload model by exploiting sequence and execution overlap information in the workload is part of our ongoing work.

2.3 Problem Statement

In this section, we first present a formulation of the database layout problem that focuses on I/O performance. We then show how to extend this formulation to incorporate important manageability and availability requirements. We define the *I/O response time* of a given statement as the total elapsed time spent performing I/O to execute that statement. Our goal is to automatically choose a database layout that minimizes the (weighted) sum of the I/O response time over all statements in the workload. We denote the I/O response time of a query Q for a given valid database layout L by *Cost* (Q, L). The formal statement of the database layout problem is shown in Figure 2. We now discuss how manageability and

availability requirements in database layout can be incorporated in our formulation. We model these requirements as additional *constraints* in the problem formulation described in Figure 2.

Given: A set of database objects $\{R_1,\ldots R_n\}$, a workload W (set of queries/updates with associated weights), and a set of disk drives $\{D_1, \ldots D_m\}$
Find: A *valid* database layout with the smallest total estimated I/O response time for the workload, i.e., find a valid layout L such that for any valid layout L' $\Sigma_{Q \in W} w_Q$. Cost (Q, L) $\leq \Sigma_{Q \in W} w_Q$. Cost (Q, L')

Figure 2: The Database Layout Problem

2.3.1. Manageability Requirements. DBAs often use a filegroup for manageability reasons as a unit of backup and restore. For example, a DBA may want to backup a set of frequently updated tables more often, and may want this set of tables to belong to a single filegroup. We incorporate such a specification by adding a *co-location constraint Co-Located* (R_i, R_k) to the definition of a *valid* layout. *Co-Located* (R_i, R_k) means that objects R_i and R_k must be placed in the same filegroup, i.e., we need to ensure that both R_i and R_k are assigned to exactly the same set of disk drives. Semantically, *Co-Located* (R_i, R_k) can be expressed as follows:

$$\forall j \in [1\ldots m]\ (x_{ij} = 0 \Leftrightarrow x_{kj} = 0)$$

A second manageability requirement arises from the fact that while DBAs may occasionally be willing to completely re-design the current database layout, in many common situations (e.g., adding an index, adding a disk drive) they would prefer an *incremental* solution. One way to incorporate such incrementality into our problem formulation is to introduce a constraint that that limits the total amount of data movement required for transforming the current database layout to the proposed layout. We note that these constraints can affect the nature of the optimization problem itself, and hence the solution to the problem as well.

2.3.2. Availability Requirements. When different disk drives have different availability characteristics – e.g., some disk drives are RAID 1 (Mirroring), others are RAID 5 (Parity), and still others are RAID 0 (no availability), the DBA may want to specify an availability constraint *Avail-Requirement* (R_i) that enforces a specific degree of availability for object R_i. For example, the DBA may want *Mirroring* for a particular critical table. Once again, we can incorporate availability requirements in the problem formulation by introducing additional constraints to the validity of a layout. Semantically, *Avail-Requirement* (R_i) can be expressed as:

$$\forall j \in [1\ldots m]\ (x_{ij} > 0 \Rightarrow AVAIL_j = A)$$

3. Architecture of Solution

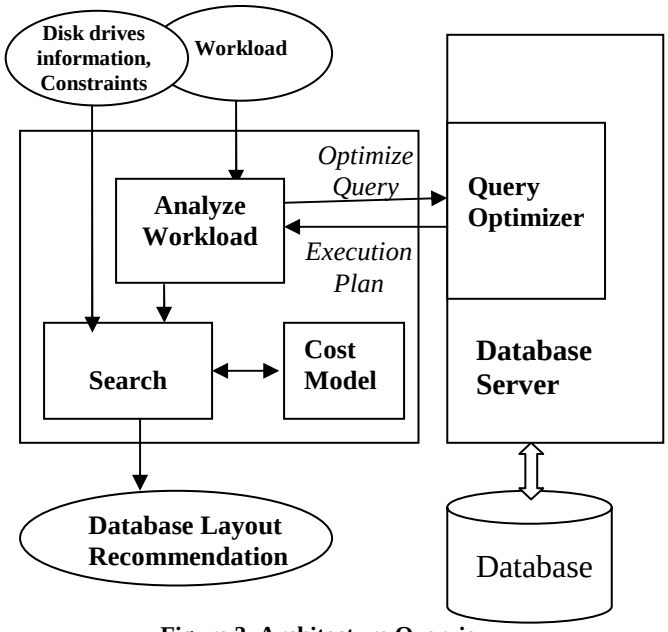

Figure 3. Architecture Overview

The architecture of our solution to the database layout problem is shown in Figure 3. We take as input the following information: (1) A database that consists of a set of tables as well as a set of other physical design objects such as indexes and materialized views. The database has a current database layout, which can be inferred by looking up the database system catalogs. (2) A workload file consisting of a set of SQL DML statements that execute against the given database. Each statement in the workload may (optionally) have associated with it a weight that denotes the importance of that statement in the workload. (3) A file containing a list of disk drives with the associated disk characteristics. The disk drives listed in this file need *not* be existing disk drives. (4) Optionally, manageability and availability constraints (as discussed in Section 2.3) that the DBA may wish to impose on the solution.

We produce as output a recommendation for the database layout that is appropriate for the given database, workload, disk drives and specified constraints. Along with this recommendation, we include an estimate of the percentage improvement in I/O response time if the recommended layout were to be actually implemented. These estimates are based on our Cost Model of I/O response time (Section 5). A novel aspect of our solution is the manner in which we exploit information about the workload to guide the choice of an appropriate layout. In our architecture, the Analyze Workload component (described in Section 4) is a preprocessing step executed prior to solving the optimization problem, that extracts information about which objects (e.g., tables, indexes) are

accessed during the execution of the workload. The Analyze Workload component is efficient since it does not actually execute the workload. Instead, it examines the *execution plan* that is generated by the query optimizer for the statement. This information is passed into the Search component, which uses the information to guide its strategy for solving the optimization problem.

The goal of the Search component is, to enumerate over the space of possible database layouts that satisfy the specified constraints, and choose the one that has the lowest total I/O response time for the given workload. The Search component relies on the Cost Model component to provide accurate information about the estimated I/O response time for the workload. The optimization problem is provably hard (Section 6.1), and thus the key challenge is to design an efficient and scalable search algorithm that ensures good quality recommendations in practice. Since the cost model may be invoked many times by the search algorithm, the scalability of the solution relies on the cost model being computationally efficient. In particular, the cost model estimates the I/O response time for a layout, without physically materializing the layout or actually executing the workload. We now describe each of the components of our solution in detail.

4. Analyzing Workload

There are two key aspects of the workload that affect the choice of database layout. The first is information about which objects are accessed during execution of the workload and total number of blocks accessed for each object. The second aspect is which sets of objects are co-accessed during execution, and the total number of blocks co-accessed. For the rest of the paper we refer to the above information as *workload information*. In this section, we describe: (a) How we represent workload information (Section 4.1) and (b) How such workload information can be extracted from a given workload (Section 4.2).

4.1 Representing Workload Information

We represent workload information in the form of a weighted undirected graph that we refer to as the *access graph* (denoted in this paper by **G**). Each node *u* in the access graph represents an object in the database. A node *u* has a weight N_u, equal to the total number of blocks of that object that is referenced during the execution of all statements in the workload. An edge exists between two objects *u* and *v* if there are one or more statements in the workload such that *both u* and *v* are co-accessed during the execution of that statement. The weight of the edge between *u* and *v* (denoted by $N_{u,v}$) is the sum over all statements in the workload of the total number of blocks of *u* and *v* that are co-accessed during the execution of the workload. Note that the access graph depends on the actual execution plan of the statements in the workload. The following example shows the access graph for a workload consisting of two queries:

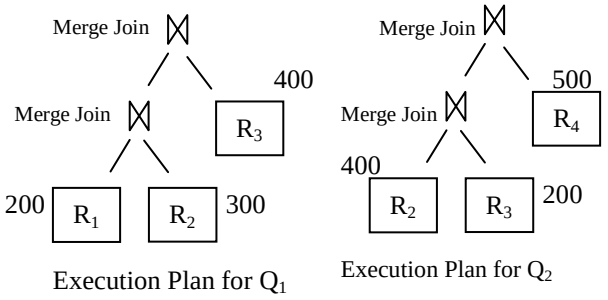

Figure 4: Execution Plans for Q_1 and Q_2

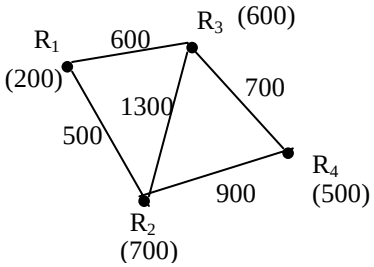

Figure 5: Access Graph for $\{Q_1, Q_2\}$

Example 2: Suppose the workload consists of two queries with the execution plans shown in Figure 4. Q_1 simultaneously accesses objects R_1, R_2 and R_3, and Q_2 simultaneously accesses objects R_2, R_3 and R_4. The total number of blocks of each object accessed in each query is also shown in the plans. Figure 5 shows the access graph for the workload $\{Q_1, Q_2\}$. The value in parenthesis on each node represents the node weight. The value on the edge represents the edge weight. For example, the edge between R_2 and R_3 shows that a total of 1300 (= 700 for Q_1 + 600 for Q_2) blocks of R_1 and R_2 are co-accessed in the workload. ♦

Finally, we note that rather than keeping information over *all subsets* of objects that are co-accessed, the access graph only keeps *pair wise* information. However, we have found in our experiments that this simplification does not significantly affect the quality of the final solution.

4.2 Extracting Workload Information

We extract workload information by analyzing the execution plan of each statement in the workload. We do *not* need to execute a statement in order to examine the execution plan of the statement. Most modern database systems provide functionality to submit a statement in a "no-execute" mode in which the query is optimized but

not executed. For example, in Microsoft SQL Server 2000 and IBM DB2, the Showplan option and EXPLAIN mode respectively provide this functionality. We note that our strategy of extracting workload information from the execution plan is *not* sensitive to the current database layout since today's query optimizers ignore the current database layout when determining a plan.

There are two important observations that guide the process of extracting information from a given execution plan. First, simply because two objects appear in the same plan does not imply that they will be co-accessed during the execution of the statement. The reason for this is that in many cases, there are *blocking* operators in the execution plan that ensure that access to one object does not begin until another object is completely accessed. We refer to the maximal subtree in the execution plan that does not contain any blocking operators as a *non-blocking subplan*. The example below highlights this point.

Example 3: Consider Q_5 of TPC-H benchmark. The query references 6 tables: *nation*, *region*, *customer*, *orders*, *lineitem* and *supplier*. Thus, without looking any further we could assume that all relations are co-accessed during query execution. However, in the actual execution plan for Q_5, which is a left-deep join tree, the tables {*nation*, *region*, *customer*, *orders*} are co-accessed and similarly {*lineitem*, *supplier*} are co-accessed, but no pair of tables across these two sets is co-accessed. This is due to the fact that after *nation*, *region*, *customer*, and *orders* are joined, there is a blocking sort operator that appears prior to the join with *lineitem* and *supplier*.♦

Second, even if an object is accessed in the execution plan, the total number of blocks of that object accessed may be significantly different than the total size of that object. The following example illustrates this point:

Example 4: Consider an execution plan that involves an index seek that retrieves RIDs of the records matching the filter condition(s), and the records corresponding to those RIDs are then retrieved from the table. Note that the number of blocks of the table accessed in the plan can, and usually does differ from the total number of blocks of the table. This number is determined by the selectivity of the predicate(s) for which the index seek is being performed, and whether the index is clustered or non-clustered. Thus, the access graph should reflect this number, rather than the total size of the table. ♦

Based on the above observations, our method for constructing the access graph from a given execution plan is described in Figure 6. Our method first decomposes the execution plan into *sub-plans*, each of which consists only of non-blocking (i.e., pipelined) operators. This decomposition is achieved by introducing a "cut" in the execution plan at each blocking operator. Next, for each database object accessed in a sub-plan, we determine the total number of blocks (say B) of that object accessed and increment the node value for that object in **G** by B. The total number of blocks of an object that is accessed can be determined based on the query optimizer's estimate of the number of rows accessed and the estimated average size of each row (available from the execution plan). For each pair of distinct objects in the sub-plan, we add an edge to **G** (if such an edge is not already present) and increment the weight of the edge by the sum of the number of blocks of both objects.

Input: Workload **W**
Output: Access graph G for **W**

1. Initialize **G** to have one node for each object in the database, and set the node value of each node to 0
2. For each statement $Q \in W$, obtain execution plan P_Q.
3. For each object R accessed in P_Q increment the node value for object R in **G** by total number of blocks of R accessed in P_Q
4. For each *non-blocking subplan* S in P_Q
5. Introduce an edge, if one does not exist, in **G** between each pair of distinct objects accessed in S. Increment the weight of the edge by the sum of the number of blocks of the two objects that define the edge.

Figure 6. Algorithm for constructing the access graph.

5. Cost Model

Our goal is to find a database layout that minimizes the total I/O response time over all statements in the workload (see Section 2.3). Any search method that solves this problem will therefore need to compute the I/O response time of statements in the workload for different database layouts. It is, however, not feasible to compute the total I/O response time for the workload by actually altering the database layout and executing statements. Thus, we instead rely on a *cost model* that *estimates* the I/O response time for a given statement Q and database layout L, without physically altering the layout or executing the query. In this section, we describe the cost model that we have adopted. Note that it is not possible to use the query optimizer's cost estimates for this purpose, because today's query optimizers are insensitive to database layout.

An effective cost model must satisfy two properties: (1) *Accuracy* – the error incurred in estimating the I/O response time should be as small as possible. Although accuracy in absolute terms is desirable, in general, it is difficult to accurately model the complex behavior of modern disk drives that perform prefetching, I/O reordering etc. Thus, similar to a query optimizer in a RDBMS, in which the goal is to accurately model the relative costs across different execution plans for a given query, our goal to accurately model the relative I/O

response time of a given query across different database layouts. (2) *Efficiency* – the computational overhead of each invocation of the cost model should be small, since the cost model may be invoked many times by the search method.

For a given layout L and a given query Q the cost model estimates the I/O response time, denoted by Cost (Q,L). When the objects required for answering the query are distributed over more than one disk drive, we define Cost (Q,L) as the I/O response time on the disk drive with the *largest* I/O response time for that query; i.e., the last disk drive to complete I/O for that query determines the I/O response time for the query. Note that the actual execution time of the query is, in general, different from the I/O response time for that query, and also depends on the CPU time taken by the query.

Input: Execution Plan P_Q for SQL statement Q, Layout L
Output: Estimated I/O response time for Q assuming layout L

1. Cost = 0
2. **For** each non-blocking sub-plan P of P_Q
3. MaxCost = 0
4. **For** each disk drive Dj
5. TransferCost = $\Sigma_i\ x_{ij}.\ B(|R_i|,P) / T_j$, where $B(|R_i|,P)$ is the number of blocks of R_i accessed in P, T_j is the read or write transfer rate (as appropriate) of disk drive D_j and the summation is taken over an object R_i if and only if R_i is accessed in P
6. Let *k* be the number of objects on D_j accessed in P.
7. **If** $k > 1$ **Then** SeekCost = $k.\ S_j.\ \min_i\ (x_{ij}.\ B(|R_i|,P))$, where Sj is the average seek time, and the minimum is taken over all objects R_i that are accessed in P
8. **Else** SeekCost = 0 **End If**
9. **If** (TransferCost + SeekCost) > MaxCost **Then**
10. MaxCost = (TransferCost + SeekCost)
11. **End If**
12. **End For**
13. Cost += MaxCost
14. **End For**
15. **Return** Cost

Figure 7: Cost Model

Our cost model assumes conventional magnetic disk technology. We model the time to service an I/O request as consisting of two parts: *seek time* and *transfer time*. We define the seek time to include the time to position the disk arm onto the appropriate cylinder and bring the appropriate sector on the cylinder under the head. The transfer time is the time taken to read (or write) the requested data once the arm and the head are in the appropriate position, and is inversely proportional to the average read (or write) transfer *rate*. The average transfer rate can be determined using any disk calibration tool or from the disk manufacturer specifications. Finally, we note that the read and write transfer rates are typically different.

Figure 7 presents the pseudocode for our cost model. We model the transfer time (Step 5) on a particular disk as the time taken to transfer all blocks accessed by the query on that disk. The seek time (Steps 6-8) on a disk drive is modeled by assuming that *on average* all objects that are co-accessed on a disk drive (i.e., in a given non-blocking sub-plan) are accessed at a rate proportional to the number of blocks accessed of each object. For example, if on a given disk drive, 10 blocks of object A and 20 blocks of object B are co-accessed, then we predict that on average, after accessing each block of A, a seek is necessary to access two blocks of B, followed by another seek to access one more block of A etc. Such a model is reasonable for most binary relational operators such as Nested Loops Join and Merge Join, as well as plans involving index seek followed by table lookup.

We now illustrate how the cost model works through the following example.

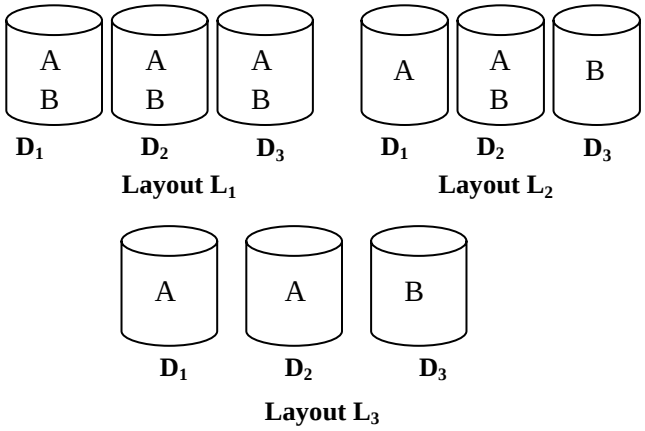

Figure 8. Cost model example.

Example 5. Consider the query "SELECT * FROM A, B WHERE A.a=B.b". Assume we find from the execution plan of this query that the object A (consisting of 300 blocks) and the object B (consisting of 150 blocks) are scanned together (e.g., in a Merge Join operator). Assume we have three identical disk drives D_1, D_2, D_3 with transfer rate T and average seek time S. Consider the layout L_1 (full striping) shown in Figure 8, in which each object is allocated on all three disk drives. Assuming equal distribution, each disk drive contains 100 blocks of A and 50 blocks of B. Thus the estimated transfer time on each disk drive is (100+50)/T and the estimated seek time is (2·50·S) for a total estimated I/O response time of (150/T + 100·S). In contrast, in layout L_2, D_1 and D_2 each

contain 150 blocks of A, and D_2 and D_3 each contain 75 blocks of B. Thus D_2 is the bottleneck disk drive for the query and its total estimated I/O response time is $(150+75)/T + 2 \cdot 75 \cdot S = (225/T + 150 \cdot S)$. In layout L_3 however, D_1 and D_2 each contain 150 blocks of A, and D_3 contains 150 blocks of B. Since there is no seek time on any of the disks and all disks contain the same number of blocks to be accessed by the query, the total estimated I/O response time of the query is 150/T. Therefore, for the above query, layout L_3 is better than layout L_1, which in turn is better than layout L_2. ♦

Our cost model is an analytical model, and thus it sidesteps the need to physically alter the database layout and actually execute queries. We present an experimental validation of our cost model in Section 7.

6. Search Strategy

In this section we present our algorithm for solving the database layout problem, i.e., finding a valid database layout having minimum total estimated I/O response time for the given workload. A popular solution to the database layout problem is *full striping*, where each object is allocated on all available disk drives (we refer to this solution in our experiments as **FULL STRIPING**)[1]. The advantage of full striping is that: (a) The method is simple to understand and manage (b) for each statement in the workload the I/O parallelism for each object accessed in the statement is maximized. However, this solution ignores the additional random I/O accesses incurred due to co-access of objects in queries, and can therefore under perform significantly.

Our search strategy uses the cost model described in Section 5 for estimating the I/O response time of the workload for a given database layout is determined. We begin by showing that for this cost model, the database layout problem is provably hard. Thus we do not expect to find a polynomial time algorithm that solves the problem optimally. Moreover, we note that the objective function we are trying to optimize, i.e., Cost (Q, L) is non-linear. Thus, rather than using generic search techniques for solving non-linear optimization problems, which tend to be computationally expensive, we try to leverage domain knowledge to develop a scalable heuristic solution.

6.1 Hardness of Database Layout Problem

Claim: The decision version of the database layout problem presented in Section 2.3 is NP-Complete when Cost (Q,L) is defined by the cost model described in Section 5.

Proof: Omitted due to lack of space. The reduction is from the Partition problem [7]. ♦

6.2 Two-Step Greedy Enumeration

We describe a two-step (heuristic) search method for the database layout problem (Section 2.3). This algorithm focuses on the performance aspect, and does not describe the modifications necessary for handling manageability and availability constraints. We omit these extensions due to lack of space. The intuition behind this method is as follows: the first step obtains an initial (valid) database layout that attempts to *minimize the co-location* of objects that are co-accessed in the workload; and the second step improves the initial solution by attempting to *increase the I/O parallelism* of objects in a greedy manner. We refer to this method in our experiments as **TS-GREEDY**. We now describe each of the two steps in more detail.

Input: Workload **W,** Access graph **G**, ***k***
Output: Database layout **L**

1. Partition nodes in **G** into *m* partitions using a graph partitioning algorithm so as to maximize the sum of edge weights across partitions.
2. **For** each partition P in descending order of total node weight
3. Assign objects in P to the smallest set of disk drive(s) ordered by decreasing transfer rate that can (a) hold the objects in the partition (b) Is disjoint from the disk drives to which previous partitions have been assigned. If a disjoint set of disk drive(s) does not exist, find a previously assigned partition P' such that sum of edge weights between P and P' is smallest, and assign P to same set of disk drives as P'.
4. **End For**
5. Let L be the layout obtained at end of Step 4, and let $C = \Sigma_{Q \in W} w_Q$. Cost(Q, L) // L is the starting layout for the greedy step
6. For each object, consider all layouts derived from L by adding at most ***k*** remaining disk drives to the object. For each layout considered, the object is allocated across the chosen disk drives in ratio of the transfer rate of chosen disk drives.
7. Of all layouts explored in Step 6, let L' be the layout with the smallest value of $C' = \Sigma_{Q \in W} w_Q$. Cost(Q, L')
8. **If** C'<C, **Then** L = L'; C = C' ; Goto 6 **End If**
9. **Return** L

Figure 9. Two-Step Greedy Search Algorithm

[1] To ensure a fair comparison with our search method, we assume that the fraction of each object allocated to a disk is proportional to the transfer rate of that disk.

Recall that the *access graph* (see Section 4.1) captures the co-access information of objects in the workload. Each edge (u,v) in the co-access graph represents the total number of blocks of objects u and v that are co-accessed in the workload. The first step (Steps 1-4 in Figure 9), which aims to minimize the amount of co-location of objects that are co-accessed in the workload is, in fact, the problem of partitioning the nodes of the access graph into a given number of partitions ($\boldsymbol{p}$) such that the sum of the weights of edges that go across partitions (i.e., the total weight of the edge *cut set*) is maximized. Intuitively, each partition contains objects that are rarely or never co-accessed together. The above problem is in fact the well known *graph partitioning* problem, which itself is known to be NP-Complete [7]. Fortunately, the graph partitioning problem has been well studied since it has many applications, and there are efficient heuristic solutions to the problem; and we use one such algorithm in our solution, namely the Kernighan-Lin algorithm [9]. We allocate objects in a partition on to the same disk drive(s). One issue in using a graph partitioning algorithm is deciding the value of $\boldsymbol{p}$ to use, i.e., how many partitions to create. Since increasing $\boldsymbol{p}$ beyond the number of available disk drives (m) cannot further reduce the co-location of co-accessed objects, we set $\boldsymbol{p} = m$.

The second step (Steps 5-8 in Figure 9), which proceeds iteratively, improves the solution obtained in the first step by attempting to increase parallelism of objects. In each iteration, we try to increase parallelism of each object by at most $\boldsymbol{k}$ (a parameter to the algorithm) additional disk drives on which the object is not already allocated. Intuitively, the parameter $\boldsymbol{k}$ controls how exhaustive this step of the search is. At the end of the iteration, the layout that reduces the cost of the workload the most is chosen as the starting point for the next iteration. The algorithm terminates when it encounters an iteration in which a layout with lower cost of the workload is not found.

Note that for an object which has little or no co-access with other objects, the greedy strategy will eventually allocate sufficient (possibly all) disk drives and will thereby achieve good parallelism for that object (similar to FULL STRIPING). However, due to its greedy nature, it is possible that the algorithm will get stuck in a local minimum. This is because when the number of disk drives on which two co-accessed objects are co-located goes from 0 to 1, the cost of the query can increase significantly (due to increased seek cost), but as the number of disk drives on which the objects are co-located increases beyond 1, the cost can decrease (below the cost for the no overlap case). Despite the above potential shortcoming, in our experiments on real and synthetic workloads (see Section 7), we have found that TS-GREEDY finds very good solutions (i.e., comparable to exhaustive enumeration in most cases) even when $\boldsymbol{k} = 1$.

The running time of the first step, i.e., the Kernighan-Lin algorithm on a graph G = (V,E) is O (|E| log |E|). Thus, in the worst case, when the number of edges in the access graph is O (n^2), the running time of the first step is O(n^2.log(n)), where n is the number of objects in the access graph. The greedy step runs in time O ($\cdot m^k n$), and thus the overall algorithm runs in time O($\cdot m^{k+1} n^2$ + n^2.log(n)). In our experiments, we use $\boldsymbol{k}$=1 and thus the running time of the algorithm is O($\cdot m^2 n^2$ + n^2.log(n)). In Section 7 we present an experimental comparison of TS-GREEDY to FULL STRIPING for different databases and workloads.

7. Experiments

We have implemented the techniques described in this paper and evaluated their quality and scalability on Microsoft SQL Server 2000. In this section, we demonstrate through our experiments that:

- The cost model (Section 5) provides good *relative* estimations of I/O response time for different database layouts.
- The greedy search algorithm (**TS-GREEDY**) presented in Section 6.2 recommends significantly better layouts than full striping (**FULL STRIPING**) in almost all the test workloads/databases.
- The running time of TS-GREEDY scales quadratically with the number of disks and number of database objects.

7.1 Experimental Setup

The experiments were conducted on a 1 GHz 256 MB Intel Pentium III machine running Microsoft SQL Server 2000 on Windows 2000 Server. The machine has 8 external disks with an aggregate capacity of 48 GB. The characteristics of the disks viz. average transfer rates and seek times were gathered using the Ziff Davis Media WinBench calibration tool [18]. The differences between the fastest and slowest disks are about 30% for both the average transfer rate and seek times. Although our problem formulation allows incorporating temporary objects, our current implementation does not support this, and therefore for fair comparison, in all our experiments we placed *tempdb* on a separate (9^{th}) disk drive.

Databases: We used the following databases: (1) TPCH1G, a 1GB TPC-H database [15]. (2) APB database [3] with about 250MB data and 40 tables. (3) SALES, an internal database that tracks the sales of the products within the company, is about 5GB in size and has 50 tables.

Workloads: The workloads we use in the experiments are summarized in Table 1. We use two benchmarks, TPC-H and APB as a part of our experiments. Both these benchmarks have complex queries that reference multiple

tables and have aggregations. We also use SALES-45, a workload that is used to analyze the sales data in the company; the queries in SALES-45 reference 8 tables on average. WK-CTRL workloads were generated to study the specific aspects of the scheme viz. the validation of cost model itself. These workloads have a small number of queries; the queries have count (*) aggregate and access almost all the table data, here *lineitem, orders, partsupp* and *part* tables in TPC-H schema.

Name	#queries	Remarks
TPCH-22	22	Standard TPC-H benchmark
SALES-45	45	Real-world workload on SALES database
APB-800	800	Workload on APB database
WK-SCALE (N)	N=100 to 3200 queries	Workloads of increasing size on TPCH1G
WK-CTRL1	5	Workloads of two table joins on TPCH1G with a simple aggregation.
WK-CTRL2	10	Mix of single table and multi-table queries, with a simple aggregation.

Table 1: Summary of workloads

7.2 Experimental Results

Validation of Cost Model: In this experiment, we validate the cost model described in Section 5. We use the following workloads: TPCH-22 (the original benchmark), WK-CTRL1 and WK-CTRL2 (see Table 1) on the TPCH1G database. For each workload, we *estimate* the cost of a layout using our cost model and then actually execute the workload after materializing the layout. For execution times, we used the average of three cold runs.

Queries	Execution Improvement	Estimated Improvement
Query 3	44%	54%
Query 9	30%	40%
Query 10	36%	51%
Query 12	32%	55%
Query 18	16%	31%
Query 21	40%	9%
TPCH-22	25%	20%

Table 2: Estimated vs. Actual for TPCH-22 queries compared to full striping

In the first part of the experiment, we analyze the behavior of individual TPCH-22 queries. We use a layout where *lineitem* is on 5 disks and *orders* is allocated on 3 disks and are completely separated; all other tables are striped across all 8 disks. Table 2 shows the improvement of the above layout in actual execution times and estimated I/O response times compared to FULL STRIPING. For Q_3, in which the I/O response time for accessing the objects is about 90% of the execution time, we get good estimation. We observe similar behavior for queries 9, 10, 12 and 18 where cost of accessing *lineitem* and *orders* accounts for most of the query cost. For query 21, we get a relatively poor approximation. This is because *lineitem* is used multiple times in the query and reflects the shortcoming of the cost model in capturing effects of buffering.

In the second part of the experiment, we generated 4 layouts, where in each case, the layout of all the TPCH1G tables is determined at random. We also generated 5 controlled layouts with different degrees of overlap between the *lineitem* and *orders* tables, as well as the FULL STRIPING layout (for a total of 10 different layouts). We used the following workloads: (a) WK-CTRL-1, (b) WK-CTRL-2 (c) TPCH-22 and (d) Five synthetically generated workloads with 25 queries each on TPCH1G with varying selection and join conditions, Group By and Order By clauses. For each workload and layout combination, we computed the cost as predicted by the cost model and then measured the actual execution time. For *each pair of layouts* we order them first by estimated cost and then by actual execution times for a given workload. We observe that the order in execution is matched by the cost model in 82% of these cases. On analyzing some of the cases where the cost model fails to identify the right order, we found those workloads to have large I/O time accessing temporary objects (e.g., ORDER BY/GROUP BY operations on large number of rows). This is because in our implementation of the cost model, we did not factor in the I/O times of temporary objects.

Effectiveness of TS-GREEDY: First we compare the estimated quality of the TS-GREEDY search strategy to FULL STRIPING. Figure 10 shows the comparison for different workloads. Note that for controlled workloads viz. WK-CTRL1 and WK-CTRL2, the estimated improvement is more than 25% higher compared to FULL STRIPING; this results from recommendations where the tables – *lineitem* and *orders* are placed on separate disks. For TPCH-22, TS-GREEDY recommends a layout where *lineitem* and *orders* are separated (5 disks for *lineitem* and 3 for *orders*) and so are *partsupp* and *part* (5 disks for *partsupp* and 3 for *part*). This separation causes the respective joins between these tables to be faster, however the individual table scans becomes slightly slower (about 5% slower for table scans on an average) as the I/O parallelism per table is reduced. The overall estimated improvement is about 20%. On materializing the above layout, we observe actual improvement of about 25%. We observe that on the SALES database, the estimated improvement is about 38%. The queries in the SALES database involve multi-table joins – TS-GREEDY separates the two largest tables in the database on 4 disks each; these tables are joined in almost all the queries. The

results remained similar even the number of disks is increased for SALES up to 64. For APB-800, the TS-GREEDY scheme recommends the same layout as FULL STRIPING. This is not surprising since the database has two large tables and several small tables; however no queries co-access the two large tables. This experiment demonstrates that TS-GREEDY can significantly outperform FULL STRIPING for workloads in which large objects are co-accessed.

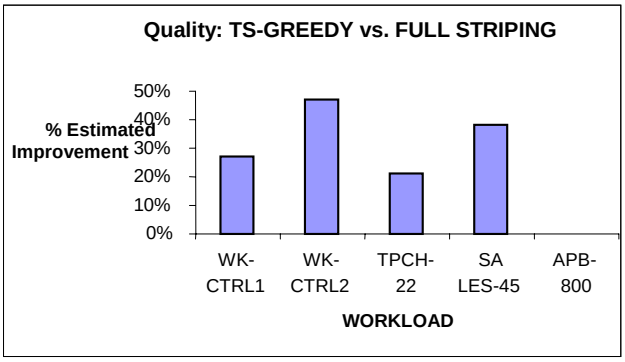

Figure 10: Comparing quality of TS-GREEDY to FULL STRIPING

Scalability of TS-GREEDY: In this experiment we demonstrate the scalability of TS-GREEDY with respect to (a) number of disks and (b) number of objects in database. For (a), we use all 3 databases viz. TPCH-22 for TPCH1G, APB-800 for APB and SALES-45 for SALES. Figure 11 shows the running time of TS-GREEDY as the number of disks are varied from 4 to 64 (doubled in every step). We plot the ratio of the running time as compared to the running time for 4 disks. The figure shows that the increase in running time is slightly more than quadratic (about 6 times as the number of disks is doubled). This is in line with our expectation from analysis of running time of the algorithm (see Section 6.2). Adding more disks also increases the evaluation time for each layout explored; and that accounts for "more than quadratic" increase in TS-GREEDY running time.

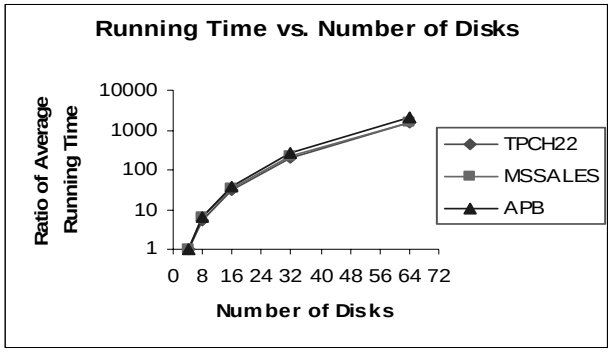

Figure 11: Running Time of TS-GREEDY vs. Number of Disks.

In the second part of experiment we study the variation of the running time of TS-GREEDY as the number of database objects is increased. We generate TPCH1G-*N*, versions of TPCH1G database where N is the number of copies of all the tables in database TPCH1G; this allows us to vary the number of objects in database. For example, TPCH1G-2 has 2 copies of all the tables in the database. We fix the number of disks to 8 and use N = 1, 2, 3, 4, 5 and 6. We generate workloads for TPCH1G-*N* as follows: we generate TPCH-88-*N*, all with 88 queries using the *qgen* query generation program of the TPC-H benchmark [15]. Next we randomly replace table names in a query with one of the *N* copies of table names using a program. This allows us to have almost identical workloads for the experiment. Figure 12 shows the running time of TS-GREEDY compared to the time taken for N=1. We observe that the running time of the greedy scheme is quadratic – it increases about 40 times when N=6. Based on these experiments, we expect that that TS-GREEDY scales well up to a few hundred disks and database objects.

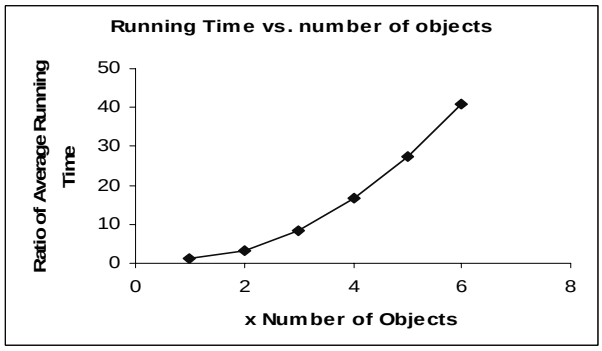

Figure 12: Running Time of TS-GREEDY vs. Number of Objects.

8. Related Work

There are three key distinguishing features of our work that compared to previous work. First, we exploit workload information at the level of SQL query execution plans. In previous work, workload information is either specified in terms of the average I/O request rate and average I/O request size per file in the system [2,14]. Second, to the best of our knowledge, ours is the first paper to exploit knowledge of co-access of objects in determining an appropriate layout. Third, unlike the work in [2,14] in which the goal is to minimize the average time of an I/O request executing against the system, our goal is to minimize the total I/O response time for a given workload.

There has been a significant amount of work in the area of storage administration and management. Early work in the HP AutoRAID project [17] demonstrated how the problem of configuring disk arrays could be made simpler by automatically moving data between two different RAID levels (RAID 1 vs. RAID 5) depending on I/O access patterns. More recently, the Minerva paper [2]

addresses the problem of not only determining which disk arrays (RAID 1/0 or RAID 5) to place the objects on based on workload information, but also looks at the capacity planning issues, i.e., what is the minimum number of disk drives that would support the throughput requirements of the workload. Similar to our approach, they also develop a cost model for predicting impact of different layouts on the workload.

The work in [14] studies a related problem – assuming files are always striped across all available disks, what should be the striping unit, i.e., the granularity at which the files should be striped. The paper also explores the issue of dynamic load balancing similar to [17], where disk bottlenecks due to hot data on a given disk are reduced by moving blocks to other disks. The issues studied in this paper are complementary to our work. We note that tissue of dynamic load balancing has also been explored in several other studies, including [1,4,6, 10,11,16]. These above studies are in contrast to our "offline" approach where we are interested in achieving a good static layout for a (given) fixed workload.

In [12], the issue of whether to cluster a file (relation) on a single disk (or as few disks as possible) or to decluster it across all disks is explored via a simulation study. The clustering option is similar to the first step of our solution (see Section 6.2) where we try to minimize co-access in the initial layout by placing an object on as few disk drives as possible. However, this may not be the best solution since it potentially gives up I/O parallelism that is explored in our second (greedy) step. In [9], it is argued that striping data across all disks is not appropriate for OLTP workloads. They propose a scheme in which only the parity data is striped and the database objects themselves are not necessarily striped across all disks.

Another area of related work [5,13] is studying how to decluster a single table across a set of disks for the case of grid-queries. The goal of these papers is to maximize I/O parallelism for the above restricted class of queries. Unlike these papers in our work (a) co-access of objects is a significant issue that affects the optimization problem and hence the search strategies, and (b) queries are more general, i.e., arbitrary SQL.

9. Summary

In this paper we present a framework for addressing the problem of assigning database objects to disk drives to optimize the I/O performance of the workload, while incorporating manageability and availability requirements. We show that exploiting knowledge of co-access of database objects is important in achieving a database layout with better I/O performance as compared to full striping. An important area of future work is extending the cost model to capture effect of concurrent execution of statements in the workload.

10. Acknowledgments

We thank Jim Gray for his comments on the paper, and Gautam Das for help with the NP-Completeness proof.

11. References

[1] AutoAdmin Project at Microsoft Research. http://research.microsoft.com/dmx/AutoAdmin

[2] Alvarez G., Borowsky E., Go S., Romer T., Szendy R., Golding R., Merchant A., Spasojevic M., Veitch A., and Wilkes J. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 2001.

[3] APB-1 *Olap Benchmark Release II*. OLAP Council Nov 1998. http://www.olapcouncil.org

[4] Copeland G., Alexander W., Boughter E., and Keller T., Data Placement in Bubba. *Proceedings of the SIGMOD 1988*.

[5] Chen C., Cheng C. From Discrepancy to Declustering: Near Optimal multidimensional declustering strategies for range queries. *Proceedings of PODS 2002*, pages 29-38.

[6] Dewan H., Hernandez M., Mok K., Stolfo S. Predictive Dynamic Load Balancing of Parallel Hash-Joins Over Heterogeneous Processors in the Presence of Data Skew. In Proceedings of PDIS, pp. 40--49, 1994.

[7] Garey M., and Johnson D. Computers and Intractability. A Guide to Theory of NP-Completeness. W.H. Freeman and Company, New York, 1979.

[8] Gray J., Horst B., Walkter M. Parity Striping of Disc Arrays. Proceedings of VLDB 1990.

[9] Kernighan B., Lin S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal, 1970*.

[10] Lee M., Kitsuregawa M., Ooi B., Tan K., Mondal A. Towards Self-Tuning Data Placement in Parallel Database Systems. *Proceedings of the SIGMOD 2000*.

[11] Lee L., Scheuermann P., Vingralek R**.** File Assignment in Parallel I/O Systems with Minimal Variance of Service Time. *IEEE Transactions on Computers, 1998*.

[12] Livny M., Khoshafan S., Boral H. Multi-Disk Management Algorithms. *Proceedings of the SIGMOD 1987*.

[13] Prabhakar S., Ghaffar K., Agrawal D., and Abbadi A. Cyclic Allocation of Two-Dimensional Data. In *Proceedings of the 14th International Conference on Data Engineering*, ICDE 1998, pages 94-101.

[14] Data Partitioning and Load Balancing in Parallel Disk Systems. Technical Report, In: The VLDB Journal Vol 7(1), Springer-Verlag, 1998

[15] TPC Benchmark H. Decision Support. http://www.tpc.org

[16] Vingralek R., Breitbart Y., Weikum G. SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load. Technical Report. In: Distributed and Parallel Databases Vol 6(2), Kluwer Academic Publishers, 1998.

[17] Wilkes J., Golding R., Staelin C., and Sullivan T.. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems* 14 (1):108-136, February 1996.

[18] Ziff Davis Media WinBench 99 Version 2.0. http://www.etestinglabs.com/benchmarks/winbench