The VLDB Journal (2003) 12: 89–101 / Digital Object Identifier (DOI) 10.1007/s00778-003-0093-1

A case for fractured mirrors

Ravishankar Ramamurthy, David J. DeWitt, Qi Su

Department of Computer Sciences, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, USA;
e-mail: {ravi,dewitt,qi}@cs.wisc.edu

Edited by P. Bernstein. Received: September 3, 2002 / Accepted: February 15, 2003
Published online: July 10, 2003 – © Springer-Verlag 2003

Abstract. The decomposition storage model (DSM) vertically partitions all attributes of a table and has excellent I/O behavior when the number of attributes accessed by a query is small. It also has a better cache footprint than the standard storage model (NSM) used by most database systems. However, DSM incurs a high cost in reconstructing the original tuple from its partitions. We first revisit some of the performance problems associated with DSM and suggest a simple indexing strategy and compare different reconstruction algorithms. Then we propose a new mirroring scheme, termed fractured mirrors, using both NSM and DSM models. This scheme combines the best aspects of both models, along with the added benefit of mirroring to better serve an ad hoc query workload. A prototype system has been built using the Shore storage manager, and performance is evaluated using queries from the TPC-H workload.

Keywords: Data placement – Disk mirroring – Vertical partitioning

1 Introduction

A number of the fundamental assumptions upon which the current generation of database systems are based have changed dramatically over the past decade. CPU speeds are improving rapidly (recently even faster than Moore's law would have predicted), and the amount of main memory that is affordable is also increasing. While disk capacities have also shown similar improvements, disk seek times and effective transfer rates (transfer rate /capacity) have improved at a much slower rate (almost by a factor of 10 slower). In addition, it appears that disk capacities are growing faster than database sizes; even the benefits of using parallelism are likely to diminish. Hence disk I/O will certainly constitute the primary performance bottleneck. Moreover, in modern architectures, cache performance has also been shown to be an important factor in the CPU time of query execution [14]. Hence database storage architectures that are more conscious of disk-arm optimizations and cache effects during query processing are needed.

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4
A5	B5	C5

ID	A
1	A1
2	A2
3	A3
4	A4
5	A5

ID	B
1	B1
2	B2
3	B3
4	B4
5	B5

ID	C
1	C1
2	C2
3	C3
4	C4
5	C5

Fig. 1. Alternate storage models

Database systems usually store all the attributes of a relation together. This format is, however, not ideal for modern database architectures given that cache misses form an important component of query execution time [5]. An alternate storage model, the decomposition storage model (DSM), uses vertically partitioned tables [11]. In this representation, each attribute of a relation is stored as a separate relation along with a surrogate that identifies the original tuple that the attribute came from.

Figure 1 shows a sample relation in the NSM representation on the far left and the corresponding DSM representation on the right. As described in [11], the DSM model maintains two copies of each partition, one clustered on IDs as shown above and the second clustered on the attribute value, which serves as an index. DSM seems to have good I/O behavior when the number of attributes touched by a query is low. Consider a sample scenario in which a selection operation has low projectivity and low selectivity, i.e., only a few attributes are projected from a large percentage of the tuples. With the DSM representation, only the partitions required by the query would be scanned, minimizing the number of disk I/Os performed while maximizing L1 and L2 data cache performance. With the NSM representation, since the query predicate is not very selective, an index would not be useful and the entire relation would be scanned. In addition, NSM would have poor cache performance [5]. PAX is a recently proposed alternative implementation of the NSM representation that employs vertical partitioning within each page [4]. For this example query, PAX would have a much better cache footprint than NSM while having the same I/O characteristics as NSM. Hence for this query it seems that DSM is the best choice. However, it is just as easy to come up with examples where the NSM representation is better. While DSM seems to be ideal for selections with low projectivity and low selectivity, as the projectivity increases, the cost of reconstructing the original tuple from the partitions begins to dominate the execution time. On the other hand, the NSM model is tuned for workloads that are highly selective and uses most of the attributes. Hence, neither storage format is optimal for all queries. The paper proposes a storage architecture that is a variant of the existing mirroring technique as a first step toward addressing this problem.

Mirroring [6,18] (RAID 1) is a technique for providing fault tolerance that maintains two (or more) identical copies of each disk. If one disk of the mirrored pair fails, the system can continue operating while the failed disk is replaced and then recovered from the mirror. Mirroring can be implemented in either hardware or software. In addition to providing fault tolerance, mirrors can also be used to improve performance by partitioning random seeks across the mirrored pair [6]. This can be critical since random seeks are very slow.

In this paper, we propose a new form of mirroring termed fractured mirrors. With this scheme, instead of the two disks in the mirrored pair being physically identical, they are logically identical. The naive implementation of fractured mirrors would store the NSM copy of a table on one disk of the mirrored pair and the DSM copy on the second disk.[1]

This scheme retains the advantages of both the NSM and DSM representations. Queries touching only a few attributes of a large number of rows will use the DSM copy. Highly selective queries or queries requiring a majority of the attributes will use the NSM copy. This idea builds on the idea of disk shadowing [6,18], which demonstrated that mirrors could be used profitably during query processing and not only for the purposes of fault tolerance. By storing the mirrors in different storage formats, we can formulate query plans that can truly maximize disk utilization while minimizing the number of L1 and L2 cache misses. However, a naive implementation of DSM can lead to surprisingly bad performance, even when only two or three attributes are accessed. The next section details the performance limitations of the DSM representation and how simple storage schemes and scan algorithms can improve its limitations. Section 3 describes our prototype, strategies for structuring the fractured mirrors, and alternative query execution plans that the use of fractured mirrors can provide. An experimental evaluation of the prototype using queries from the TPC-H benchmark suite follows. The paper concludes with a presentation of related work and strategies for handling updates.

2 Storing and scanning DSM partitions

2.1 The naive implementation

The straightforward way of implementing the vertical partitions of the DSM model is to store each vertical partition as a separate relation with two attributes – an integer that acts as an identifier and the column's value as illustrated previously. When used to store the Line-item table from the 1-GB version of the TPC-H benchmark (which has 16 attributes), this approach has very poor space utilization and performance. With the NSM representation, this table occupies about 1.1 GB and a full table scan takes 74.5 s.[2] The DSM representation, on the other hand, occupies 2.8 GB. For DSM, the attributes were assembled one tuple at a time like a traditional scan. Table 1 illustrates how a naive implementation of DSM provides a performance advantage only when a single attribute from the table is touched.

Table 1. Performance of DSM

Projectivity	Scan time (s)
1	68.29
2	138.06
4	366.86
8	759.39

Several factors contribute to the poor performance of this implementation strategy. First, the naive DSM implementation stores each (ID, AttrValue) pair as a standard database record on a slotted page [19]. While the slot overhead is generally not significant when a record is used to hold a tuple in the NSM representation, it can become significant when the record is used for a single (ID, AttrValue) pair. Furthermore, for fixed-length attributes, whose position on the page can be computed from the length of the attribute and the ID, the slotted page representation is redundant. The second significant source of wasted space is the ID itself. Storing a 32-bit or 64-bit identifier with each attribute can easily double the space required to store a table. It is very important to keep in mind that the important issue is not the disk space consumed by the slot array entry or the ID as disk space is essentially free these days. The issue is that the extra space can significantly increase the number of disk I/O operations that must be performed when the partition is accessed.

Another drawback of the naive strategy is that it is not possible to quickly reassemble a tuple from its vertical partitions given the tuple's ID. Some form of index such as a B-tree mapping ID to the attribute value is required to do this efficiently. This leads to an alternative representation in which each vertical partition is stored as a B-tree on ID with the leaf pages containing (ID, AttrValue) pairs. While this approach still wastes space storing a tuple's ID once for each attribute value plus incurs the cost of a slot array entry, it makes the task of reassembling a tuple given its ID straightforward. More importantly, it leads us to a refined representation that we describe below.

2.2 A sparse B-tree-based representation

Our refined design uses a modified B-tree design in which the overhead of the redundant IDs is eliminated for both fixed-length and variable-length attributes and the slot array overhead is eliminated for fixed-length attributes.

Our approach is based on several simple observations. First, IDs are system generated by incrementing a counter and are never reused.[3] Thus, new (ID, AttrValue) pairs are always appended to the rightmost leaf node of the B-tree. In

[1] If a table is horizontally striped across multiple mirrored pairs, the rows stored on a single disk will be stored in their NSM representation on one disk and their DSM represenation on the second.

[2] The NSM copy and DSM partitioned were stored as files in Shore configured to have a 128-MB buffer pool and 32-KB page size.

[3] The handling of deletes is discussed in a later section.

addition, for fixed-length attributes, given the ID of the lowest attribute value on a leaf page, there is no need to store the IDs of the remaining attributes as they can be computed given the attribute's offset from the start of the page. This avoids the need for either a slot array or IDs. For variable-length attributes, a standard slot array is necessary, but the attribute's position in the slot array can be used to calculate the attribute's ID. Consequently, for fixed-length attributes, the B-tree leaf pages contain only attribute values without IDs or slot arrays, raising the effective space utilization to essentially 100%. The upper levels of the B-tree are organized in a normal fashion with the key entry for a leaf page containing the ID value corresponding to the smallest tuple on the leaf page.

It is to be noted that this representation can be further optimized for fixed-length records. If storage extents are guaranteed to be contiguous, a DSM partition consisting of fixed-length records can be stored as a sequential file and individual records can be accessed using simple offset computation on physical RIDs [17]. In this case, a B-tree index would not be required. However, in this paper we use the sparse B-tree for storing both fixed-length and variable-length records; this is because Shore [9] (which is used as the storage manager for the experiments) does not guarantee contiguous extents. In any case, this would provide a lower bound on performance. A storage scheme that avoids the overhead of the B-tree for fixed-length records should provide better performance.

Processing the attribute values in a DSM partition happens in one of two ways. For a sequential scan of all values, the index is first traversed to the leftmost leaf and then the leaf pages are scanned sequentially. To retrieve the attribute value for the tuple with a particular ID, the B-tree is traversed to locate the correct leaf page by searching for the index entry that covers the ID. An index entry is said to cover a particular ID if the ID lies between the index entry and its succeeding entry in the index. Once the correct leaf page is located, the page is read and the offset computation described above is used to locate the desired attribute value.

2.3 Tuple reconstruction algorithms for DSM

Scan is a fundamental database operation that scans all tuples in a table, possibly applying one or more predicates in the process. When one or more of a table's attributes are not required by subsequent operators in the query, the scan is normally combined with a project operator to eliminate unwanted attributes as output tuples are produced. In a database system that uses the NSM (or PAX) storage representation, the scan operation is trivial to implement; successive pages of a relation are read until the end of the file is reached. In the case of the B-tree DSM representation described in Sect. 2.2, several different scan algorithms are possible. In this section, we describe and compare these algorithms for reconstructing a tuple (or portions of a tuple) from the B-trees used to hold the vertical partitions.

2.3.1 Page-at-a-time reconstruction

The simplest DSM reconstruction algorithm begins by opening a sequential scan on the B-trees of each attribute required to produce the output relation plus those attributes on which a predicate is to be applied. The scans are processed in lock-step one tuple at a time, any applicable predicates are applied, and qualifying tuples are materialized in their NSM representation on the reconstruction operator's output stream. The primary disadvantage of this approach is that it incurs a random seek each time a new B-tree leaf page is read.

2.3.2 Chunk-based reconstruction

A scan of a relation stored in the DSM representation can also be viewed as a multiway join of each of the table's vertical partitions. Since today's database systems include very efficient join algorithms, one might be tempted to simply use the standard join code to reconstruct a table from its partitions. However, reassembling a table of 20 attributes with a 19-way join is likely to overwhelm any database system. The join of the DSM partitions is actually a very special kind of merge-join in which the input tables are already sorted on the join attribute (i.e., the virtual ID value) and each attribute value joins exactly one attribute from all the other partitions and thus is handled exactly once.

If N pages of memory are available and K attributes are accessed by the scan, the reconstruct-in-chunks algorithm begins by dividing the memory into K chunks of size N/K pages. Each chunk corresponds to one attribute. It then opens scans on the B-trees of each of the K attributes, filling each of the K chunks with N/K leaf pages from the corresponding B-tree before proceeding to the next chunk. The value of this simple tactic cannot be overemphasized. While disk capacity increased by a factor of 1000 in the 20-year period from 1980 to 2000 (80 MB to 80 GB), the time for a random disk seek has decreased by only a factor of 6 (from 30 ms to 4.9 ms) over the same period. Filling each vertical partition a chunk at a time reduces the number of random seeks performed by the join by a factor of N/K. The other key technique the algorithm uses is to process attribute values in a chunk in cache-line size units to insure that the L1 and L2 data caches are used as effectively as possible. Thus, with a 64-byte cache line and 4-byte attribute values, the algorithm constructs 16 output tuples at a time.

2.3.3 Performance

This section evaluates the effectiveness of the suggested storage schemes and scan algorithms. We first show how to select an appropriate value for N/K – the number of pages to use for each attribute with the chunk-based merge algorithm.

The graph shown in Fig. 2 illustrates how the reconstruction time using the chunk-based merge algorithm varies as a function of the chunk size for scanning 5 and 10 attributes from the 1-GB version of the TPC-H Line-item table. As the graph illustrates, beyond about six pages no further improvement occurs. For all subsequent experiments, a chunk size of five pages is used.

The next graph (Fig. 3) shows the DSM scan times as a function of the number of attributes being reassembled using the page-at-a-time and the chunk-based merge algorithms. For reference, the NSM scan time for the table is also shown. While the page-at-a-time algorithm can reassemble only four

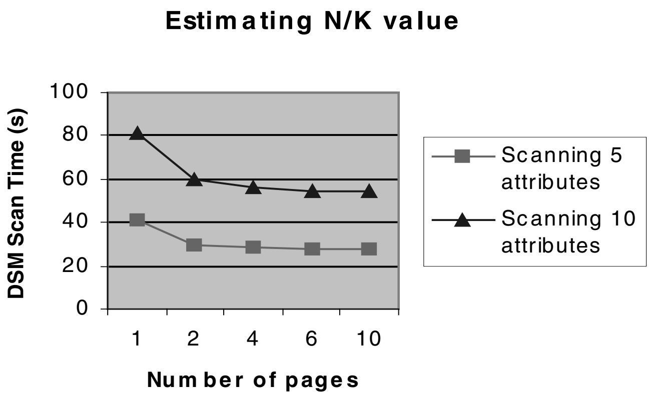

Fig. 2. Estimating N/K value

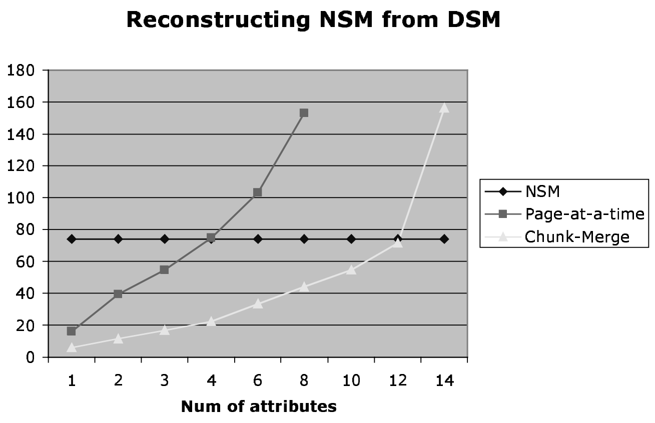

Fig. 3. Reconstructing NSM from DSM

attributes in less time than the time required to sequentially scan the entire NSM table, the chunk-based algorithm can reassemble 12 out of 16 attributes before its performance becomes worse. The results for both algorithms are much better than the results presented for the naive DSM implementation in Sect. 2.1, which required 138 s to reassemble just two attributes. Since the naive representation also used a page-at-a-time algorithm, the primary difference is due to the improved B-tree-based storage scheme described in Sect. 2.2.

2.4 DSM scan optimization techniques

The task of reconstructing NSM tuples from the DSM partitions is essentially a join between the individual partitions. We next examine how traditional optimization techniques for joins such as pushing down selections and join ordering are applicable to the chunk-merge algorithm. The naive implementation of this algorithm reconstructs the NSM tuples before evaluating any of the scan's predicates. It is in fact possible to push down the selection predicates. The idea is to split any predicates on the NSM view of the table into predicates on each attribute being assembled. After an attribute chunk has been read into memory, any applicable predicate on that attribute is evaluated. This simple optimization has two side effects. First, since predicates are evaluated attribute-wise, the selection operation has excellent cache performance. Moreover, since predicates are evaluated in an eager fashion, those attributes that do not qualify need not be merged. This leads to savings in the "join-processing" overheads of the algorithm. For highly selective predicates this savings can be substantial. It follows from this optimization that "join-ordering" of the partitions should be in descending order of the selectivities of the predicates on the corresponding attributes (starting with the most selective predicate first).

For example, consider TPC-H query 6. This query computes a set of aggregates on the Line-item table (6 million tuples), accessing only four of the table's attributes. The DSM plan for this query would assemble just the required partitions using the chunk-merge algorithm and then apply the required predicates and aggregate operations. The query was evaluated with and without the push-down selection optimization described above. The results are shown below:

DSMScan (naive evaluation):
55.03 s (CPU: 50.63 I/O: 7.82)

DSMScan (with predicate push-down):
18.07 s (CPU: 12.27 I/O: 8.54)

These results clearly indicate that, for certain predicates, pushing down the selection predicates can dramatically reduce overall query execution times by eliminating unnecessary CPU operations.

We think these results are very encouraging. By eliminating the redundant storage of IDs and, by using better scan algorithms, these results indicate that the DSM representation, when implemented properly, can provide better performance over a much wider range of situations than previously believed. In the following section, we describe a new mirroring strategy that incorporates both NSM and DSM copies of a table.

3 Mirroring using DSM

3.1 Introduction

Both the NSM and DSM storage models have inherent limitations. Database systems, having to pick one, have traditionally chosen NSM, as it is more suitable for OLTP-like applications. Most database systems today employ some form of redundant storage to provide tolerance to disk failures. While RAID-5 is frequently used today, the trend is toward increased use of RAID-1 (mirroring). Even though mirroring incurs a 100% storage penalty, write operations are more efficient than RAID-5 since there is no check-sum block to be updated.

In this section, we describe a new form of mirroring that we term fractured mirrors. The basic idea is simple: rather than two disks in a mirror being identical physically, they are instead logically identical. In particular, with fractured mirrors one copy of each table is stored in an NSM representation and one is stored in a DSM representation. This section outlines how such a system can be constructed while retaining the advantages of both formats without losing the advantages of mirroring.

3.2 Data placement for fractured mirrors

The simplest way of implementing mirrors would be to put the NSM copy on one disk of the mirrored pair and the DSM copy

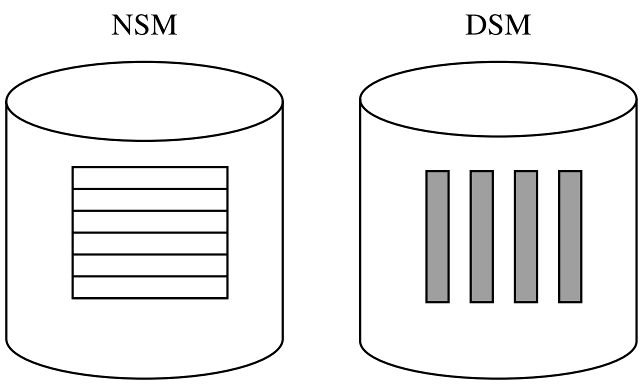

Fig. 4. Naive logical mirroring

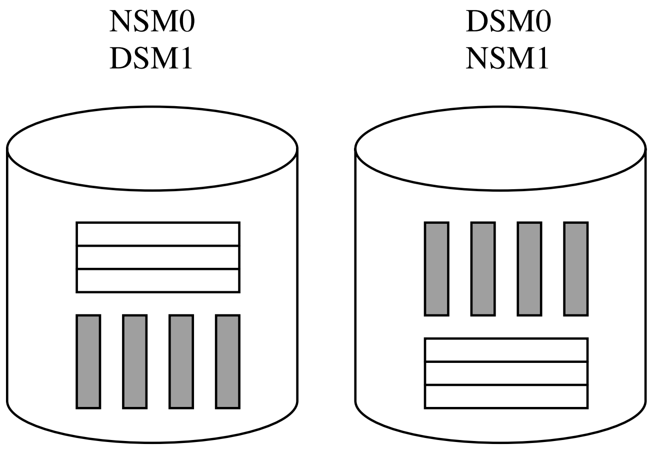

Fig. 5. Fractured mirrors

on the other disk, as shown in Fig. 4. For each query, the optimizer would decide which copy is best and the corresponding representation would be used to execute the query. The main disadvantage of this approach is that if the query workload is skewed toward one of the two representations, the two disks will not be utilized uniformly. Another problem is that random seeks cannot be distributed between the mirrors. This is because NSM and DSM do not have similar performance when it comes to index lookups. NSM can retrieve the entire tuple in one access, while DSM must retrieve the additional attributes by means of additional index lookups using the ID. Hence the load on the two disks will not be symmetric. It is, however, possible to place each storage model on hardware specifically tuned for the model. This is an idea to explore in the future.

A solution to this problem is the notion of fractured mirrors, in which data is placed on the mirrors in the following fashion. Consider a system with two disks. As shown in Fig. 5, the NSM copy is declustered across the two disks using a round-robin-based scheme into two equal-sized fragments NSM0 and NSM1. On disk 1, along with NSM0 we store the tuples of NSM1 in DSM format, and along with NSM1 on disk 2 we store the tuples of NSM0 in DSM format. Since both disks have NSM0 and NSM1 in some data format, they both have a complete copy of the data. Hence this constitutes a valid mirroring scheme. Even if the query workload is skewed toward one representation, since both storage formats are represented on each disk, accesses will be uniformly distributed across both disks. More importantly, we can now partition random seeks between disks in a symmetric fashion. Since the NSM copy is declustered, on average one half of the random page accesses will be handled by each disk, a key property that the original mirroring scheme guarantees [6].

An important issue is the choice of an appropriate declustering algorithm [16]. Any suitable partitioning algorithm can be used as long as it produces a reasonably even distribution. In some ways, fractured mirrors are similar to RAID-10, which first mirrors an entire file and then declusters blocks between mirrors for higher bandwidth. The significant difference, of course, is the use of multiple storage representations with fractured mirrors. Another fundamental difference is that the various RAID schemes are usually implemented in the disk controller. Fractured mirrors must be implemented in the database system, which may introduce some degree of inefficiency.

The data placement strategy indicated in Fig. 5 deals only with the replication of data on a single disk. If the original table(s) have been declustered across multiple disks, fractured mirrors can still be used as follows. Let *swap* define the procedure by which a system of two disks using naive mirroring can be converted into fractured mirrors. Consider a system of $2N$ disks (N disks and their mirrors). Choose any suitable data placement strategy to populate the first N disks. For every relation on disk i, store the corresponding DSM representation on disk $N + i$. This results in a mirrored system that uses the naive strategy. Then, for every pair of disks (i, $N + i$), apply the *swap* procedure to generate the fractured-mirrors version of the N disk system.

Given a query, the database system can now select the storage format most appropriate for evaluating the query. Issues in generating query plans for the mirrors are discussed in Sect. 4. In the following section, we present some experimental results executing queries from the TPC-H suite on this system.

3.3 Experiments on the TPC-H suite

A prototype relational system was built using Shore[9] as the underlying storage manager and included the normal relational operators such as scan, join, split, merge, etc., along with operators to implement functionality for the chunk algorithm. The experiments were run on a Pentium III dual-processor machine (550 MHz) with 1 GB of main memory running Linux 7.1. Three disks (sequential bandwidth 15–20 MB/s) were used for storing data: two Shore volumes were stored on the first two for the fractured mirrors, and the third disk was used to hold the Shore log file. The Shore buffer pool size was set to 128 MB. A page size of 32 KB was used. 1 GB of TPC-H data was generated using the data generator. These data were converted into a tuple representation and stored on the two volumes, as shown in Fig. 5. The queries were run and their results validated as indicated in the benchmark specification [2]. All reported times are cold times and are the average of three runs. The Shore buffer pool was flushed between queries by dismounting and remounting the disks between runs. All running times are reported in seconds. A brief description of

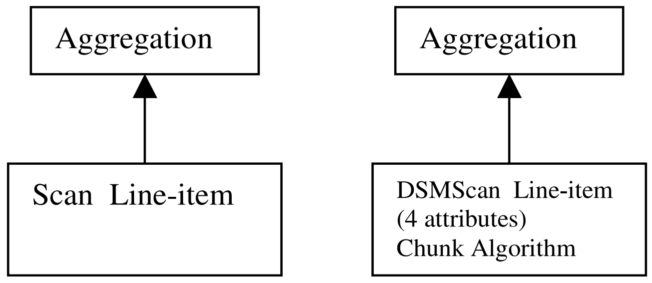

Fig. 6. TPC-H query 6

each query along with its execution times is given. Query plans are illustrated wherever appropriate.[4]

The initial queries demonstrate the advantage of maintaining a copy of the database in the DSM form. In each of these queries, the DSM plan assembles all the required partitions of a relation in a leaf node of the query plan using the chunk algorithm.

Query 6:

Query 6 computes an aggregate over selected rows of the Line-item table. The DSM plan only scans the relevant attributes. (Only four attributes are used by the query.)

Execution times (seconds):
DSM: 10.03
NSM: 75.41

Query 1: Query 1 is similar to query 6 except it contains more complicated aggregate computations. The query touches seven of the attributes from the Line-item table and has a predicate on the l_shipdate field that selects about 97% of the rows.

Execution times (seconds):
DSM:56.43
NSM:87.82

Query 12:

Query 12 is a join query between the Line-item and Orders tables followed by an aggregation. The DSM plan consists of two DSM_Scan nodes feeding into the join operator. Four attributes are used from Line-item and two are used from Orders.

Execution times (seconds):
DSM: 84.43
NSM: 240.46

Query 10:

Query 10 is a four-way join between the Line-item, Orders, Customer, and Nation tables. The query includes an order-by and a group-by clause and requires only the first 20 results. Order-by was implemented using the sort routine of Shore.

The DSM plan is shown in Fig. 7 with the number of attributes required from each relation. Again DSM has the best performance even though the query touches most of the attributes of the Customer table (seven out of nine).

[4] For simplicity, sequential plans are shown. The actual versions executed are the parallel versions taking into account the declustering in the two-disk system.

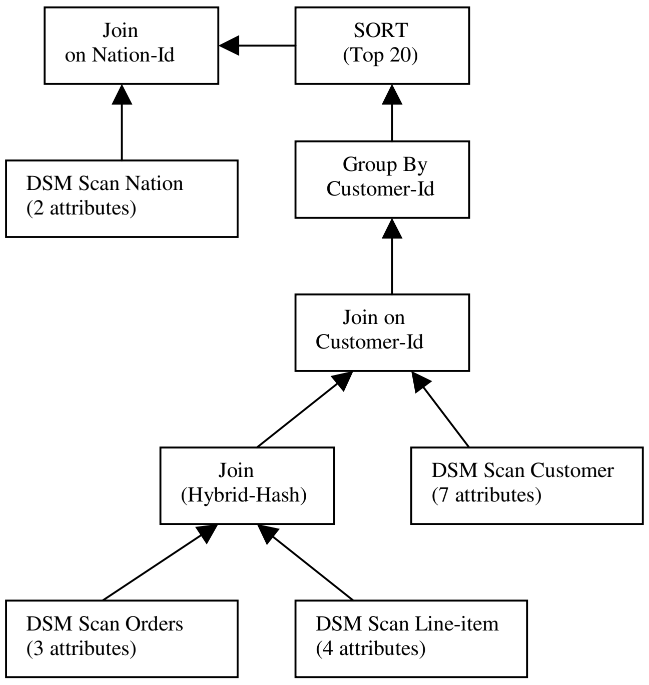

Fig. 7. TPC-H query 10

Execution times (seconds):
DSM: 257.08
NSM: 435.68

Query 1*:

This query demonstrates the advantage of having both NSM and DSM representations in a mirrored system. This query is a slightly modified version of a query in which the predicate on l_shipdate is reversed to make it highly selective. An index was built on the l_shipdate column to evaluate this query efficiently. This query shows how DSM performance deteriorates with highly selective queries with even moderate degrees of projectivity since it must probe additional indices to fetch the required attributes. In the DSM plan, the index on l_shipdate produces the IDs of the qualifying tuples and then the required attributes are obtained by using the ID to probe the corresponding sparse B-tree indices. The six attributes probed are assembled using a DSMScan, and the aggregate is evaluated. In this case, choosing the NSM plan would be better and would be feasible in the mirrored architecture. The query plans are illustrated in Fig. 8. Execution times (seconds):
DSM: 14.21
NSM: 6.76

Query 19:

The DSM plans in the previous cases assembled all the attributes of a particular relation in a leaf node. However, in some cases, it may be more efficient to put together the attributes in multiple stages based on the selectivity of each of the attributes touched by the query. Query 19 is a join between the Line-item table and the Parts table. The query computes the revenue of parts by using the extended price and discount attributes of the Line-item table for those tuples that qualify the join. It turns out that the join is highly selective, producing

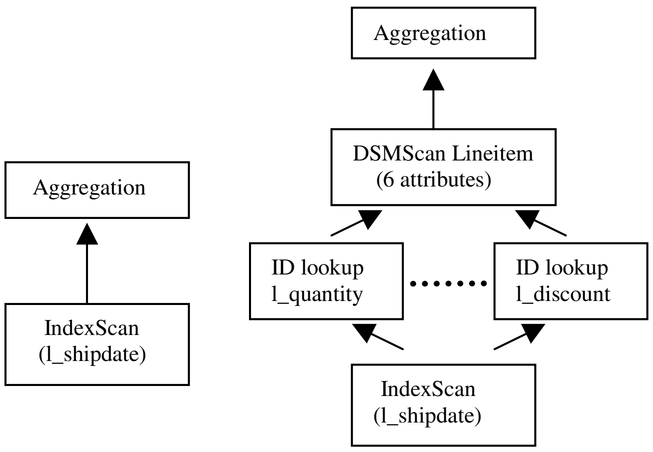

Fig. 8. TPC-H query 1*

only 121 tuples. Moreover, the attributes required for computing the aggregate are not required anywhere else in the plan. Hence with the DSM plan, instead of scanning all six required attributes from Line-item in a leaf-level operator (DSM-1), another plan would scan only four attributes at the leaf level (DSM-2). The IDs of the tuples produced by the join would then be used to probe the B-trees corresponding to the DSM partitions of the remaining two attributes that are needed to compute the aggregate. Since this algorithm will incur a large number of random accesses to DSM tuples, it is viable only for very highly selective predicates such as the one in this query (in which only 121 tuples out of 6 million satisfy the predicate). The DSMScan on line-item would produce the tuple IDs along with the attributes. The join would project the tuple IDs of the tuples that qualify the join, these IDs would be used to probe the index on the partitions l_extendedprice and l_discount, and the values will be used to compute the aggregate. The two alternative DSM query plans are shown in Fig. 9. The l_shipinstruct attribute used by this query is a fixed-length string type. Since the attribute contains only four distinct values, the string values are encoded as an integer field to exploit the fixed-length optimizations for DSM suggested earlier.

Execution times (seconds):
NSM: 273.55
DSM-1: 205.88
DSM-2: 201.50

Hybrid Plan:
We use a simple query to demonstrate the notion of hybrid plans, plans in which both data representations are used to evaluate the query plan. The query selected is a modified version of query 12, which is a join between Line-item and Orders. An additional predicate is added to the Order table to restrict the number of order tuples, and all attributes from the Orders table are projected for the tuples that qualify the join. The best plan for this query is a hybrid plan in which the NSM copy of the Orders table and the DSM copy of the Line-item table are joined.

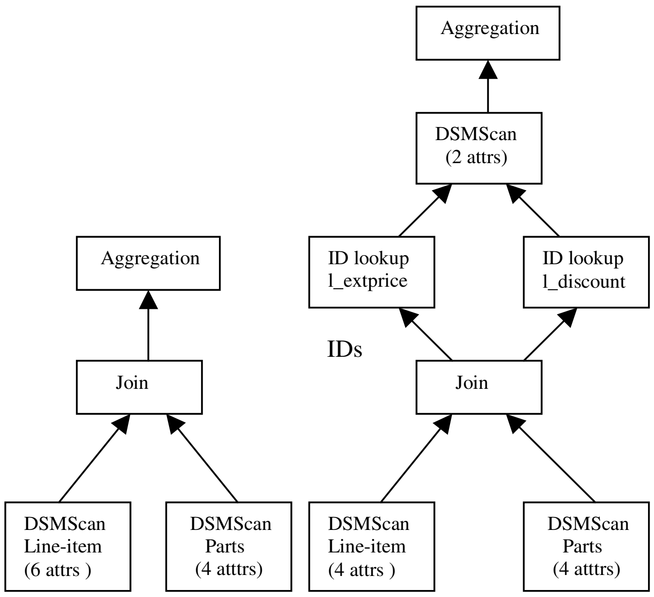

Fig. 9. Alternate DSM plans for TPC-H query 19

Table 2. Summary of results

TPC-H query	NSM/DSM ratio
Query 6	7.51
Query 1	1.56
Query 1*	0.48
Query 10	1.69
Query 12	2.86
Query 19	1.36
Hybrid Plan	1.47

Execution times (seconds):
Pure DSM: 177.07
Pure NSM: 204.37
Hybrid: 139.06

As we can see, the hybrid plan is better than both the pure NSM and pure DSM plans. Each of these plans has one leg of the join that is not optimal in terms of disk I/O. The hybrid plan uses the best means to scan each relation in the join and hence is better than the other two plans.

The speed-up obtained by using DSM for the discussed plans is summarized in Table 2. We can see that using DSM yields speed-up factors ranging from 1.3 to 7.5. We have also seen that, for some queries such as query 1*, DSM performs poorly. Having both copies as part of a mirrored system is likely to serve a wider range of query workloads. Another advantage of maintaining both representations is that the best plan for certain queries is one in which both representations are employed. It is to be noted that these numbers do not necessarily depict the best-case scenario for DSM. In an environment having relations with large numbers of attributes the speed-up factors could be much more substantial. For example, one of the key tables used for the Sloan Digital Sky Survey has over 400 attributes [22].

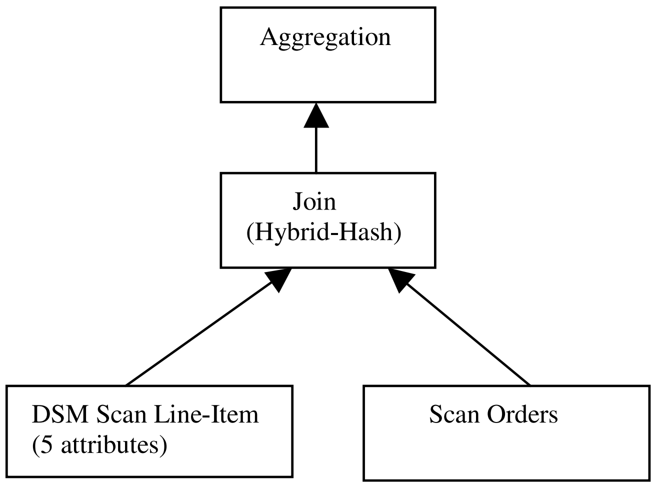

Fig. 10. Hybrid plan

3.4 Synchronizing the mirrors

Once the mirrors have been created, they must be kept synchronized through the course of database operations such as inserts, updates, and deletes. In traditional mirroring, all such operations are applied directly to both copies, which is not feasible with fractured mirrors since the DSM and NSM copies do not have identical performance characteristics under these operations. For instance, an insert operation corresponding to a tuple with n attributes would result in n insert operations on the corresponding vertical partitions of the DSM copy. Hence, given high update rates, the overhead of keeping the mirrors up to date may result in a serious performance penaltly.

The solution we are considering uses an intermediate representation of the relation to serve as a differential file to record updates and inserts [20]. The differential file is implemented as a relation with three attributes having the schema (Tuple-Id, Attribute-Id, Value). A single entry represents a new attribute value of the original tuple. An insert operation would now result in the insert of n tuples to this relation, the main difference being that the inserts can be implemented as a sequence of sequential writes since the differential file is clustered on the Tuple-Id value. With main memories becoming larger and larger, the differential file can be cached in memory until the actual updates have been applied to the appropriate DSM partitions. Once we have recorded the inserts and updates in the differential file, we must propagate these values to the original partitions regularly to ensure that the differential file does not grow too large. Eventually we hope to piggyback these writes whenever there are reads to nearby cylinders, as discussed in [21].

A side effect of these schemes is that the differential file must be consulted during query processing. The chunk-based reconstruction algorithm described in Sect. 2.3 can be extended in a simple fashion to consult the differential file and the delete bitmap while assembling tuples. The original k-way merge becomes a $k + 2$-way merge with the differential file and the delete bitmap read in tandem with the vertical partitions being assembled. Tuples from the differential file and the delete bitmap corresponding to the tuple IDs currently being reconstructed in memory are also read into main memory. We essentially ensure that the tuple being assembled has not been deleted and is also merged with the differential file updates for it before sending it to the output stream. Any additionally inserted tuples in the differential file must also be processed. The differential file and the delete bitmap are clustered on tuple ID for efficient merging during the chunk algorithm.

A disadvantage of caching the differential file in memory is that the time to reconstruct a disk after a failure may be longer than with traditional mirroring. If failure only involves a disk, the failed disk can be reconstructed using its mirror and the memory-resident differential file. If a failure involves a loss of a disk as well as the loss of memory, then it will be necessary to also use the transaction log to recovery the updates that had not yet been applied to the DSM copy on disk. The proposed scheme for handling updates should work well as long as it is possible to keep the differential file small and propagate the changes to the partitions on a regular basis.

Deletes are handled in a slightly different fashion. We maintain a single column relation. Each page of the relation contains a bitmap. For instance, a page 8 KB large would contain a bitmap with about 64,000 entries. To delete a particular tuple in the original relation, we need to find the page that contains the bit entry corresponding to the given tuple ID. The index structure described in Sect. 2.2 can provide this access path. Once the corresponding bit has been located, it is set to 0 to indicate that the tuple has been deleted. This is similar to the notion of an existential bitmap outlined in [17].

In mixed workloads, the DSM partitions will eventually contain holes corresponding to tuples that have been deleted. Unless each partition is compacted periodically, scan times will continue to increase even if the total number of tuples remains relatively constant. The obvious approach would be to lock the partitions, drop all indices, compress the partitions, and then rebuild the indices. This unfortunately would preclude using the DSM copy until the process has been completed. The NSM copy could, howeve r, still be used for answering queries. We briefly outline two techniques for reorganizing the partitions in an online fashion.

Preserve Tuple ID Mappings: In this scheme, as each tuple is written to a compacted partition, they are assigned a new tuple ID value corresponding to the logical position of the tuple. In addition, a table that relates old and new tuple ID values is constructed. With this mapping table, existing indices can continue to be used. While compaction is in progress, queries that access compacted or uncompacted partitions exclusively can be executed in the normal fashion. However, queries that access both types of partitions will require an additional processsing step in which the new tuple IDs in the compacted partitions are replaced by the appropriate old tuple ID values prior to merging the different types of partitions.

Retain Old Tuple Ids: This scheme retains the old tuple IDs when compacting. For fixed-length attributes the DSM partition would simply be rewritten without the deleted records. For variable-length attributes, the corresponding data entries will be deleted from the index structure described in Sect. 2.2, without reclaiming the slot entries (since tuple IDs are not reclaimed). An advantage of this scheme is that existing index structures can be used without modification. Moreover, queries can be answered using both compacted and uncompacted partitions since the tuple IDs in each DSM tuple are not changed. However, when a fixed-length attribute is accessed through an index, the deleted bitmap will still need to

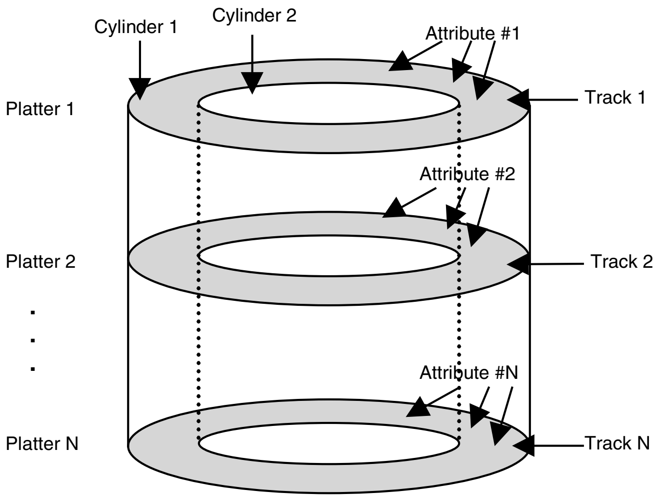

Fig. 11. Placing partitions on adjacent cylinders

be consulted. For example, if the k-th record is accessed, the bitmap must be traversed to determine its actual position in the compacted partition.

It would be an interesting study to compare these alternatives in an update intensive environment like TPC-C. It is possible in such environments that traditional mirroring (NSM + NSM) will likely perform better than fractured mirrors. However, it may be the case that, if the updates are mainly updates to individual attributes and not inserts (as is the case with TPC-C), fractured mirrors should actually perform similar to traditional mirrors.

Fractured mirrors are suitable for systems that have complex queries and a relatively small update-to-query ratio in the workload. We are currently working on an intermediate system that is likely to have update performance between the two extremes discussed. Some of the main differences of this system and the one we have discussed before are as follows. Only fixed-length records are partitioned, and all variable-length records are clustered as a single partition. By careful data placement based on disk geometry we hope to reduce seek times.

Traditional data clustering lays out data sequentially cylinder by cylinder. We modify this slightly for the partitions. The first partition will be placed on track 1 of cylinder 1, 2, 3, etc. The second partition will be placed on track 2 of cylinder 1, 2, 3, etc. Thus, when we seek to a particular cylinder, the corresponding tracks will contain the vertical partitions of a table. With a single seek operation about ten partitions can be reached on a modern disk drive. Given that all these partitions contain fixed-length attributes, the number of disk seeks needed to propagate updates or inserts to these partitions will be minimal. Clearly, the update performance in this approach will be intermediate to normal mirroring and the fractured-mirrors approach. The data placement strategy is illustrated in Fig. 11. For this increase in performance, we need to invest more effort in data placement. If the workload characteristics are known in advance and if the update rates do not merit the increased complexity of this approach, the original scheme of fractured mirrors would be more suitable.

In the future, we intend to study compression techniques for DSM in tandem with careful data placement techniques for the partitions.

4 Issues in query processing

This section outlines how queries can be evaluated for fractured mirrors. Traditionally query processing proceeds without regard to whether or not the data are mirrored. The read requests generated during query execution are appropriately scheduled between mirrors based on expected seek times by a low-level disk scheduler. In our architecture, there is an opportunity to push this decision up to the level of the query optimizer, as it can choose a plan that better exploits the semantics of the different data formats used in the mirrors. This section explains how a traditional bottom-up search-based query optimizer can be extended to generate plans in this environment.

4.1 Optimize-twice approach

Since we have data stored in two different data models, a simple way to look at query optimization is to determine the best possible way to execute the plan using the NSM and DSM representations and then pick the better of the two plans. Consider two relations R (R1, R2, R3) and S (S1, S2) and a simple join query between them (assume R1 and S1 are the attributes on which the query is joined):

Query = $\pi R2(R \bowtie S)$ projects all R2 attributes, which qualifies the join.

The corresponding DSM schema for R and S is:
R-1 (id, R1), R-2 (id, R2), R-3 (id, R3)
and S-1 (id, S1), S-2 (id, S2)

The equivalent query for the DSM relations would be $\pi R2(R-1 \bowtie R-2 \bowtie S-1)$.

The DSM query has the original join between the R1 partition and S1 partition and an additional join based on ID to retrieve the R2 attributes that belong to this join result. The chunk algorithm discussed would be implemented as a specialized join algorithm that can be used for joining partitions. The query optimizer would use standard join ordering schemes and decide the best plan for the above query using a suitable cost model. Having obtained the best plans for each storage model, we can look at the optimizer cost estimates for both the plans, and we would pick the plan having the better cost. Even though this approach is simple, each query is optimized twice, which would add to the overhead of query execution.

4.2 Combined search of plan space

Ideally we would like to explore the search space of both storage models in a combined fashion, thereby eliminating the redundancy of the optimize-twice approach. This section outlines how a bottom-up search-based optimizer can be extended to achieve this objective. A detailed overview of the bottom-up search is available in the survey [13].

We need a new logical operator *Assemble* that corresponds to an operator that joins operator trees having partitions of the same relation (based on the ID column). If we are assembling leaf nodes that are scans on vertical partitions, the algorithms discussed in Sect. 2.3 would be suitable for implementing this operator. If two trees of operators having partitions of the same original relation are being assembled, the IDs of the partitions produced by the first tree would be used to probe the sparse B-tree indices of the partitions in the second. This operator is similar to the materialize operator proposed in [7] for evaluating pointer joins in object-oriented database systems.

Consider the simple join query considered in the previous section. First consider how the search space of plans is explored for the NSM model. For simplicity, assume that the database has no indices to use for this query. The given query is $\pi R2(R \bowtie S)$. The base nodes for the search would be scan nodes on relations R and S. Let these be denoted by {R} and {S}. In the first phase of search-space exploration, all possible operators will be applied to the base set. For this example we will have a join operator that will generate the nodes {R, S} and {S, R}, corresponding to the two possible join orders. Since both of these nodes have the same logical properties, only the plan with the least cost will be retained. It happens that the plan chosen will be complete since it can implement the projection on R2, which is the only operation left. Thus, it would be chosen as the optimal plan.

Let us consider combined optimization for both NSM and DSM. When starting the search, all possible initial access paths must be added as base nodes. Hence we need to include the nodes in the previous case as well as scans on all partitions touched by the query, i.e., {R}, {S}, {R1}, {R2}, {S1}. Among the set of operators that would generate new operator trees would be the join operator (as in the previous case) and the Assemble operator, which would combine partitions.

The join operator would generate joins for nodes that can satisfy the join predicate (between attributes R1 and S1). It would generate the following nodes: {R, S}, {R1, S}, {R1, S1}, {R, S1} and the corresponding nodes with the join orders reversed. The Assemble operator would generate {R1, R2}. The Assemble operator is not sensitive to the order in which the partitions are assembled. Hence {R2, R1} will not be generated. These nodes will again be grouped into equivalence classes and the minimal cost node will be retained for each class. Here are the classes and the best plans for those classes (we do not include the nodes generated due to join commutativity for simplicity):

Class 1: {R, S} {R, S1} – best plan {R, S1}
Class 2: {R1, S} {R1, S1} – best plan {R1, S1}
Class 3: {R1,R2}

Among these classes, only Class 1 is complete since it can project the attribute R2 (the class includes a scan on all attributes of relation R).

In the next phase of optimization, the Assemble operator will combine {R1, S1} and {R1, S} with {R2} to generate {R1, S1, R2} and {R1, S, R2}. The join operator will combine {R1, R2} with {S} and {S1} to generate {R1, R2, S} and {R1, R2, S1}. Now, all the generated trees will belong to Class 1 and the best plan among {R,S1} (the current best plan) and these plans would be picked as the optimal plan.

The Assemble operator would maintain logical properties that combine the logical properties of the individual partitions being scanned. Among the logical properties we need to maintain is the notion of *attributes that come into scope*. For example, the join operator can evaluate the join {R1, S} because the attributes required for the join from R (just R1) have already come into scope. By maintaining this property we can ensure that vertical partitions are also considered in the joins and the Assemble operator will ensure that larger partitions are generated from smaller partitions. This ensures that the combined space of plans for NSM and DSM will be explored together.

An interesting benefit of this approach is that it is straightforward to generate hybrid plans in which the final query plan has part of the query using NSM and another part using DSM. For our query example, {R, S1} is a hybrid plan. In certain cases, such plans are better than fully DSM or NSM plans, which would be the only type of plans, generated in the optimize-twice approach. Thus, conventional query optimization can be extended in a simple fashion to generate plans for fractured mirrors. Such an optimizer is currently being developed, and we are in the process of investigating further details including cost models, search space efficiency, and appropriate heuristics for restricting search space.

5 DSM and index structures

The experiments in Sect. 3.3 demonstrated that the NSM storage model is not optimal for those queries that access a small number of attributes of a table. Since most commercial systems today use the NSM model, to optimize OLAP-style queries these systems resort to using index structures such as covering indices and projection indices. In this section, we briefly examine how DSM compares to these alternatives.

5.1 Covering indices

An index is termed a covering index if the index's key value includes all the attributes required by the query.[5] In this case, the query can be answered by simply scanning the covering index without accessing the base table. Thus, covering indices facilitate "index-only" plans. If the covering index key consists of exactly the attributes required by the query, no unnecessary I/O operations are performed to read unused attributes, which is exactly the advantage obtained through the use of the DSM. Even though both techniques serve similar purposes, there are subtle differences.

In general, a covering index is optimized for a particular query. Given sufficient information about the query workload, it is possible to build covering indices optimized for certain queries, and a query plan that uses the appropriate covering index is very likely to be the best plan. If no single covering index is optimized for a query, it is still possible to use multiple covering indices to answer a query. For instance, if the keys of two (or more) B-tree indices together cover the set of attributes required by a query, the query can be evaluated by

[5] In general, covering indices are implemented as multiattribute B-tree indices.

Table 3. Evaluating query 6

Alternative query evaluation plans	
Plan I	Scan (10,4,5,6)
Plan II	(10,4) Join (5,6)
Plan III	(10) Join (4,5,6)
Plan IV	Scan (10,4,5,6,11)
Plan V	(10,4,2) Join (5,6,3)

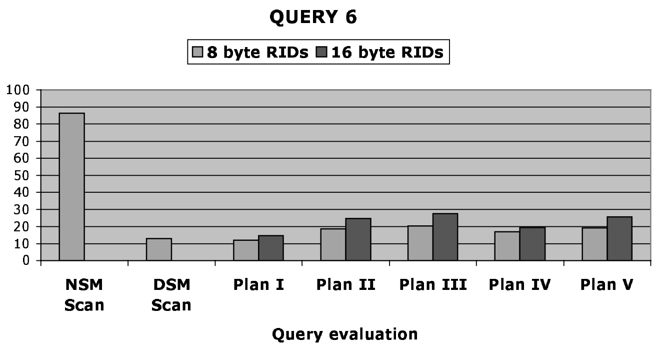

Fig. 12. Comparing covering indices and DSM

joining these indices using the RID values stored in the leaf entries of the index. DSM partitions, on the other hand, are a representation of the base data and are not optimized for any particular query. Any query can use exactly the minimal number of partitions necessary, using the chunk-merge algorithm to merge partitions at runtime.

The key difference between covering indices and DSM partitions is the fact that they are clustered differently. Covering indices are clustered on the (multi) attribute key value and thus can be used to efficiently evaluate range predicates. DSM partitions, on the other hand, are clustered on the ID value and hence can be efficiently merged using the chunk-merge algorithm but are not useful for evaluating range predicates. An important point to observe is that the chunk-merge algorithm is a ***no-write*** algorithm. Regardless of the number of partitions being merged or the size of the partitions, the algorithm never produces any intermediate files because the partitions are already clustered on the ID value. Merging multiple covering indices, on the other hand, is a join operation and will generally involve writing hash partitions or sort runs to disk unless the predicates are highly selective.

We next describe some preliminary experiments comparing different evaluation strategies for query 6 of the TPC-H benchmark. This query requires only four attributes from the Line-item table (numbers 10,4,5,6). We compare the performance obtainable with covering indices to the DSM.

Consider the different query plans listed in Table 3 for evaluating query 6. A covering index is represented by the list of attribute numbers that constitute its key. For instance, Plan I evaluates query 6 by scanning a covering index that includes all four of the attributes required by the query as part of its key. Plan II joins two covering indices, each having two of the attributes required by the query. The other plans use other combinations of covering indices. Some of the covering indices have additional attributes in order to simulate cases in which the indices may not be exactly optimized for the query at hand.

The experiments were performed using Shore as the storage manager. Covering indices were first implemented using Shore B-trees with multiattribute keys. However, since Shore does not link the leaf pages of B-tree indices, the upper levels of the index end up being traversed multiple times for range predicates that span multiple leaf pages. Even though these index pages are present in the buffer pool, the cost of locking and latching every page leads to suboptimal performance. In order to provide the fairest comparison, covering indices were simulated by storing the key values and the RID as records in a sequential file sorted on the key value. The execution times for the plans listed in the table, as well as the NSM and DSM plans, is shown in the graph below. The experiments were repeated using both 8-byte and 16-byte RIDs.

The query plans that join multiple covering indices use a hash-join algorithm in which the hash partitions fit entirely in memory. Thus, the join algorithm does not perform any disk I/O. This illustrates the best case for merging two covering indices. As the experiments indicate, Plan I using 8-byte RIDs is the best plan. The interesting point to note is that the DSM plan is better than all the remaining plans; this is due to the efficient chunk-merge algorithm for merging partitions. For certain plans, DSM can provide a speed-up factor of nearly 2.5. These experiments are, however, not meant to represent a definitive comparison between DSM and covering indices. Instead, the results do, however, clearly indicate that there are cases in which using DSM will have better performance. A key point to keep in mind is that having DSM partitions does not preclude having covering indices; DSM is a way of storing data. Additional index structures as deemed appropriate can still be built to further optimize queries.

5.2 Projection indices

A projection index is a materialization of a particular column of a relation. Maintaining projection indices on all columns of a relation is equivalent to the notion of storing the relation in DSM format. Thus, one can consider using projection indices on all columns along with the base table stored in NSM as an alternative to using fractured mirrors. Even though the two are logically equivalent, this scheme can have inferior performance for a couple of reasons. First, the chunk-based merge algorithm combined with the heuristic for pushing down predicates can give dramatic performance benefits over algorithms that do not use similar techniques while merging projection indices. Second, the projection indices are stored as part of the database; if the database is itself mirrored for fault tolerance purposes, we will end up with two copies of the tables in their NSM representation and two copies of the projection indices. This is likely to incur significant overhead on each update. Fractured mirrors constitute a single mechanism for both fault tolerance and improved I/O performance and are likely to provide superior query and update performance.

6 Related work

Disk technology trends were discussed in [12]. The authors point out that, while disk prices have dropped by a factor of

10,000, accesses per second have grown by a factor of only 100. The importance of cache performance in query processing was studied in [5,14], and PAX was proposed as a solution [4]. Given that we have an additional copy in DSM, some of the advantages of PAX can be bought by simply using the DSM copy. The effectiveness of using PAX for the NSM copy in fractured mirrors is to be studied as future work.

The notion of using DSM for good disk bandwidth and cache performance is similar to the notion of building covering indices for the query at hand. But for query workloads whose patterns are not known before hand, it may not be possible to build efficient covering indices. By using the individual DSM partitions and the chunk algorithm, we can simulate the functionality of covering indices. Covering indices have been studied in detail and are available in products like Microsoft SQL Server. The performance of the DSM model has been studied in [11,15]. The conclusions were that DSM is better when the projectivity is low and the selectivity is medium to low, while NSM is better when both the projectivity and selectivity are high. Performance of single attribute modification is the same for DSM and NSM, while NSM provides much better record insert/delete performance. The query-processing algorithm presented in [15] used the notion of join indices, while we have outlined how query optimization can be extended in a general fashion to support DSM. A performance evaluation of DSM using the TPC-D benchmark has also been carried out in [8] using the Monet main memory database system.

Some of the very early prototype database systems that hinted at using decomposed storage were [23,24]. The BUBBA project was among the better-known projects that advocated the use of DSM. The BUBBA system [10] proposed a notion of using a set of inverted files and a remainder relation as an online copy instead of mirroring. DSM has also been used as a physical storage model to implement object-oriented data models. Query-rewriting schemes for translating queries on an object-based model into DSM is presented in [8] using the Monet main memory database system. The notion of projection indices [17] is an implementation of DSM used in warehousing environments. Among today's database products, Sybase-IQ [1], whose target market is data warehousing, uses vertically partitioned attributes as its storage model. By using efficient compression techniques and advanced bitmap indexing, aggregate queries (which are typical in a warehousing environment) can be answered very efficiently. We were unable to obtain further details as to how the partitions were indexed and how queries were optimized. The fact that DSM is suitable for decision support workloads has already been discussed [1,8,17]. As far as we can tell, this paper is the first to propose the notion of mirroring using different data storage formats.

Using mirrors to optimize reads by distributing random seeks between the disks was first discussed in [6]. This technique was extended to optimize write performance. The scheme described in [18] used a notion of distorted mirrors, which worked at the granularity of disk blocks and cleverly managed the blocks in two partitions. The notion of the mirrors not being identical is similar to our general idea, though their paper is not concerned with storage models. It would be interesting to see if some of their optimizations for placing disk blocks would still be valid under the current scheme of fractured mirrors. Our proposal of propagating updates to the vertical partitions by using a differential file in memory is similar to update piggybacking suggested in [21].

The three-column relation proposed for organizing the differential file has been proposed in a different context. New e-commerce applications require data schemas that are constantly evolving and hence require table structures that are more flexible than the standard NSM representation. Agrawal et al. [3] proposed the three-column relation as the standard storage format, whereas we use it only as a differential file to record the updates.

7 Conclusions and future work

The decomposition storage model (DSM) has not found widespread acceptance by database vendors. Given technology trends and the need for storage architectures that are more aware of disk-arm and cache effects during query processing, we feel that DSM is likely to play a more significant role in future database system products. This paper identified some of the fundamental performance limitations of DSM. Contributions of this paper include alternate storage schemes and scan algorithms for DSM that provide a dramatic increase in performance over the naive implementation.

A new mirroring technique was proposed as a storage architecture that can best exploit the advantages of DSM. We would like to think of our work as extending the current spectrum of mirroring techniques. Based on the workload mix (queries and updates), the complexity of queries, and the update frequency, one can pick the mirrored architecture that is most suitable. As shown for complex queries such as those in the TPC-H suite, there are obvious benefits in maintaining a copy in DSM. For workloads that do not have high update rates, the notion of fractured mirrors is likely to suffice. For higher update rates and TPC-H-like queries, the optimized version of fractured mirrors that pays more attention to data placement is likely to be a better choice. For simple queries with high update rates, the original mirroring scheme is the best.

As part of our future work, we intend to examine a number of issues. Given a query workload, we need to decide good data placement schemes for the partitions. The current evaluation of the system has focused primarily on the TPC-H query workload. Future work would include experimenting with different transaction protocols to update the DSM copy efficiently and an evaluation of the system using an OLTP benchmark like TPC-C. Query optimization for the mirrors offers many problems to be studied. We also need efficient schemes to handle variable-length records and NULL values. Currently the mirroring scheme is implemented in software; it would be interesting to see if RAID hardware could be leveraged to any extent.

Acknowledgements. We would like to thank the referees for their detailed comments, the Borg team and Anastassia Ailamaki for sharing their code, and Joseph Burger for many clarifications on the efficient use of Shore.

References

1. SybaseIQ White Paper (2001) www.sybase.com
2. TPCH Benchmark Specification (2001) www.tpc.org
3. Agrawal R, Somani A, Xu Y (2001) Storage and querying of e-commerce data. In: Apers PMJ, Atzeni P, Ceri S, Paraboschi S, Ramamohanroa K, Snodgrass RT (eds) Proceedings of the 27th internal conference on very large databases, Rome 11–14 September 2001. Morgan Kaufmann, San Francisco, pp 149–158
4. Ailamaki A, DeWitt DJ, Hill MD, Skounakis M (2001) Weaving relations for cache performance. In: Apers PMJ, Atzeni P, Ceri S, Paraboschi S, Ramamohanroa K, Snodgrass RT (eds) Proceedings of the 27th international conference on very large databases, Rome, 11–14 September 2001. Morgan Kaufmann, San Francisco, pp 169–180
5. Ailamaki A, DeWitt DJ, Hill MD, Wood DA (1999) DBMSs on a modern processor: where does time go? In: Atkinson MP, Orlowska ME, Valduriez P, Zdonik SB, Brodie ML (eds) Proceedings of the 25th international conference on very large databases, Edinburgh, 7–10 September 1999. Morgan Kaufmann, San Francisco, pp 266–277
6. Bitton D, Gray J (1988) Disk shadowing. In: Bancilhon F, DeWitt DJ (eds) Proceedings of the 14th international conference on very large data bases, Los Angeles, 29 August–1 September 1988. Morgan Kaufmann, San Francisco, pp 331–338
7. Blakeley JA, McKenna WJ, Graefe G (1993) Experiences building the open oodb query optimizer. In: Buneman P, Jajodia S (eds) Proceedings of the 1993 ACM SIGMOD international conference on management of data, Washington, DC, 26–28 May 1993. ACM Press, New York, pp 287–296
8. Boncz PA, Wilschut AN, Kersten ML (1998) Flattening an object algebra to provide performance. In: Proceedings of the 14th international conference on data engineering, 23–27 February 1998, Orlando. IEEE Computer Society, New York, pp 568–577
9. Carey MJ, DeWitt DJ, Franklin MJ, Hall NE, McAuliffe ML, Naughton JF, Schuh DT, Solomon MH, Tan CK, Tsatalos OG, White SJ, Zwilling MJ (1994) Shoring up persistent applications. In: Proceedings of the 1994 ACM SIGMOD international conference on management of data, Minneapolis, 24–27 May 1994, pp 383–394
10. Copeland GP, Alexander W, Boughter EE, Keller TW (1988) Data placement in bubba. In: Boral H, Larson P (eds) Proceedings of the 1988 ACM SIGMOD international conference on management of data, Chicago, 1–3 June 1988. ACM Press, New York, pp 99–108
11. Copeland GP, Khoshafian S (1985) A decomposition storage model. In: Navathe SB (ed) Proceedings of the 1985 ACM SIGMOD international conference on management of data, Austin, TX, 28–31 May 1985. ACM Press, New York, pp 268–279
12. Gray J, Graefe G (1997) The 5-minute rule revisited and other storage rules of thumb. ACM Sigmod Record 26(4):63–68
13. Ioannidis YE (1996) Query optimization. ACM Comput Surv 28(1):121–123
14. Keeton K, Patterson DA, He YQ, Raphael RC, Baker WE (1998) Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In: Proceedings of the 25th annual international symposium on computer architecture, Barcelona, 27 June–1 July 1998. ACM/IEEE Computer Society, New York, pp 15–26
15. Khoshafian S, Copeland GP, Jagodis T, Boral H, Valduriez P (1987) A query processing strategy for the decomposed storage model. In: Proceedings of the 3rd international conference on data engineering, 3–5 February 1987, Los Angeles. IEEE Computer Society, New York, pp 636–643
16. Livny M, Khoshafian S, Boral H (1987) Multi-disk management algorithms. In: Proceedings of the 1987 ACM SIGMETRICS conference on measurement and modeling of computer systems, Alberta, Canada, 11–14 May 1987. ACM Press, New York, pp 69–77
17. O'Neil P, Quass D (1997) Improved query performance with variant indexes. In: Proceedings of the 1997 ACM SIGMOD international conference on management of data, Tucson, 13–15 May 1997. ACM Press, New York, pp 38–49
18. Orji CU, Solworth JA (1993) Doubly distorted mirrors. In: Buneman P, Jajodia S (eds) Proceedings of the 1993 ACM SIGMOD international conference on management of data, Washington, DC, 26–28 May 1993. ACM Press, New York, pp 307–316
19. Ramakrishnan R (1997) Database management systems. McGraw-Hill, New York
20. Severance DG, Lohman GM (1976) Differential files: their application to the maintenance of large databases. TODS 1(3):256–267
21. Solworth JA, Orji CU (1990) Write-only disk caches. In: Garcia-Molina H, Jagadish HV (eds) Proceedings of the 1990 ACM SIGMOD international conference on management of data, Atlantic City, 23–25 May 1990. ACM Press, New York, pp 123–132
22. Szalay AS, Kunszt PZ, Thakar A, Gray J, Slutz DR (2000) Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. In: Chen W, Naughton JF, Bernstein PA (eds) Proceedings of the 2000 ACM SIGMOD international conference on management of data, 16–18 May 2000, Dallas. ACM Press, New York, pp 451–462
23. Titman PJ (1974) An experimental database system using binary relations. In: Proceedings of the IFIP working conference on data base management, Corsica, France, 1–5 April 1974, pp 351–362
24. Todd S (1975) Prtv: an efficient implementation for large relational data bases. In: Proceedings of the international conference on very large data bases, 22–24 September 1975, Framingham, MA. ACM Press, New York, pp 554–556