Database-Aware Semantically-Smart Storage

Muthian Sivathanu*, Lakshmi N. Bairavasundaram†,
Andrea C. Arpaci-Dusseau†, and Remzi H. Arpaci-Dusseau†
* *Google Inc.* †*Computer Sciences Department, University of Wisconsin, Madison*

Abstract

Recent research has demonstrated the potential benefits of building storage arrays that understand the file systems above them. Such "semantically-smart" disk systems use knowledge of file system structures and operations to improve performance, availability, and even security in ways that are precluded in a traditional storage system architecture.

In this paper, we study the applicability of semantically smart disk technology underneath database management systems. For three case studies, we analyze the differences when building database-aware storage. We find that semantically-smart disk systems can be successfully applied underneath a database, but that new techniques, such as log snooping and explicit access statistics, are needed.

1 Introduction

Processing power is increasing in modern storage systems. For example, the Symmetrix storage array, a high-end RAID from EMC, contains nearly 100 processors and up to 256 GB of memory [9]. Unfortunately, the ability to leverage the computational power within traditional storage systems has been limited due to its narrow block-based interface [8, 10]. With protocols such as SCSI, storage arrays receive only the simplest of commands: read or write a given range of blocks. Hence, the storage system has no knowledge of how it is being used, *e.g.*, whether two blocks are a part of the same file, or even whether a given block is live or dead.

To bridge this information gap, recent research has proposed the idea of a *semantically smart disk system* [30] that either learns of or is embedded with knowledge of the file system using it. This semantic information within the storage system allows vendors to build more functional, reliable, higher-performing, and secure storage systems. For example, by exploiting knowledge of file and directory structures, a storage system can deliver improved data availability under failure [29].

Previous research on semantically smart disk systems [3, 28, 29, 30] has assumed that a commodity file system (*e.g.*, Linux ext2, Linux ext3, NetBSD FFS, Windows FAT, or Windows NTFS) is interacting with the disk. In this paper, we explore techniques for semantically-smart disk systems to operate beneath database management systems (DBMS). Given that database systems form a significant and important group of clients of storage systems, we would like to see if the benefits of semantically smart storage can be applied to this realm.

Whether operating beneath a file system or a database, a semantically smart disk system performs similar operations, such as tracking which file or table a particular block has been allocated to. However, a DBMS tracks different information and organizes its data on disk differently than a file system does. For example, most file systems record within each file's metadata certain statistics, such as the most recent access and modified time. Given that a DBMS is more specialized, it does not track these general statistics. Second, in file system workloads, the directory structure tends to be a reasonable approximation of semantic groupings; that is, users place related files together in a single directory. However, in a DBMS, the semantic grouping across different tables and their indexes is dynamic, depending upon the query workload.

Our general finding is that these differences are fundamental enough to require changes for semantically smart storage. To build database-aware storage, we investigate two techniques that were not required for file systems. First, we explore *log snooping*, in which the storage system observes the write-ahead log (WAL) records written by the DBMS; by monitoring this log, the storage system can observe every operation performed by the DBMS before its effect reaches the disk. Second, we explore the benefits of having the DBMS explicitly *gather access statistics* and write these statistics to storage. We find that it is relatively simple to add these statistics to a DBMS.

To investigate database-aware storage, we implement and analyze three case studies that have been found to work well underneath file systems. First, we study how to improve storage system availability with D-GRAID, a RAID system that degrades gracefully under failure [29]. Second, we implement a DBMS-specialized version of FADED, a storage system that guarantees that data is unrecoverable once the user has deleted it [28]. Finally, we explore how to improve second-level storage-array cache

*Work done while at the University of Wisconsin-Madison hit rates with a technique known as X-RAY [3].

Our experience indicates that semantically-smart disks can work well underneath database systems. In some cases, database systems are a better fit than file systems for semantically-smart storage, such as for secure delete [28]. In this case, the presence of the transactional semantics in a DBMS allows the disk to more accurately track dynamic information. As a result, functionality that requires absolutely correct inferences can be implemented without changing the DBMS; in contrast, this same functionality required changes to the file system. However, for two of the case studies, D-GRAID and X-RAY, we find that a DBMS does not supply all of the desired access information to the storage system. As a result, better results are obtained if we slightly modify the DBMS.

The rest of this paper is organized as follows. In Section 2, we review related work in database-aware storage and discuss the advantages and disadvantages of semantically-smart disks. In Section 3, we describe the general techniques needed for a semantic disk to extract information from the DBMS. In Sections 4, 5, and 6, we present our case studies. Finally, we discuss the range of useful techniques in Section 7 and conclude in Section 8.

2 Background

Placing more intelligence in disk systems to help database systems has come in and out of favor over the years. For a summary of work in this area, see Keeton's dissertation [17], page 162. One of the earliest examples is the idea of "logic per track" devices proposed in 1970 [31]; for example, given a disk with some computational ability per head, a natural application is to filter data before it passes through the rest of the system.

Later, the idea of database-specific machines was refuted, for example in 1983 by Boral and Dewitt [5]. The primary reason for the failure of such approaches was that they often required non-commodity components and were outperformed as technology moved ahead; worse, database vendors did not wish to rewrite their substantial code base to take advantage of specific features offered by certain specialized architectures.

However, as processing power has become faster and cheaper, the idea of "active disks" has came into focus once more. Recent work includes that by Acharya *et al.* [1] and Riedel *et al.* [25]; in both efforts, portions of applications are downloaded to disks, thus tailoring the disk to the currently running program. Much of this research focuses on exactly how to partition applications across host and disk CPUs to minimize data transferred.

In contrast to much of this previous work, the semantically-smart approach does not require specialized hardware components or sophisticated programming environments [3, 28, 29, 30]. High-end storage arrays are a good match for this technology, as they often have multiple processors and vast quantities of memory. However, building semantic knowledge of higher-level systems into a storage array has both benefits and drawbacks.

The main benefit of the semantic-disk approach is that it increases functionality; placing high-level semantic knowledge within the storage system enables new systems that require both the low level control available within the storage array, and high level knowledge about the DBMS; such systems are precluded in traditional storage architectures. For example, previous research has shown that semantic disks can improve performance with better layout and caching [3, 30], can improve reliability [29], and can provide additional security guarantees [28].

However, the semantically-smart approach also leads to a few concerns. One concern is that too much processing will be required within the disk system. However, many researchers have noted that the trend is of increasing intelligence in disk systems [1, 25]. Indeed, modern storage arrays already exhibit the fruits of Moore's Law and the EMC Symmetrix storage server can be configured with up to 100 processors and 256 GB of RAM [9]. These resources are not idle, but nonetheless hint at the relative simplicity of adding more intelligence.

A second concern is that placing semantic knowledge within the disk system ties the disk system too intimately to the file system or DBMS above. For example, if the DBMS on-disk structure changes, the storage system may have to change as well. In file systems, on-disk formats rarely change; for example, the format of the ext2 file system has not significantly changed in its 10 years of existence [30], and current modifications take great pains to preserve full backwards compatibility with older versions of the file system [33]. In the case of a DBMS, format changes are more of a concern. To gain some insight on how often a storage vendor would have to deliver "firmware" updates in order to keep pace with DBMS-level changes, we studied the development of Postgres [24] looking for times in its revision history when a dump/restore was required to migrate to the new version. We found that a dump/restore was needed every 9 months on average, more frequent than we expected. However, in commercial databases that store terabytes of data, requiring a dump/restore to migrate is less tolerable to users; indeed, more recent versions of Oracle go to great lengths to avoid on-disk format changes.

A final concern is that the storage system must have semantic knowledge of each layer, whether a file system or a DBMS, that could possibly run upon it. Fortunately, there are only a few file systems and database systems that would need to be supported to cover a large fraction of the market. Further, much of the functionality in a semantic disk is independent of the layer above; thus, only a small portion of the code needs to handle issues that are specific to each file system or DBMS. Finally, if a storage vendor wants to reduce the burden of supporting many different database platforms, they can target a single important database (*e.g.*, Oracle) and just provide standard RAID functionality for other systems. Interestingly, high-end RAID systems already perform a bare minimum of semantically-smart behavior. For example, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple sector writes) reaches disk atomically [6]. In summary, storage vendors are already willing to commit resources to support database technology.

3 Database-Aware Techniques

To implement powerful functionality, a storage system can leverage higher-level semantic information about the file system or DBMS that is running on top. In this section, we describe the types of information a semantic disk requires underneath a DBMS, and discuss how such information can be acquired. Database-specific semantic information can be broadly categorized into two types: static and dynamic.

Since our experience has primarily been with the Predator DBMS [27] built upon the SHORE storage manager [19], we illustrate our techniques with specific examples from Predator; however, we believe the techniques are general across other database systems.

3.1 Static information

Static information is comprised of facts about the database that do not change while the database is running. The storage system can obtain static information either by having such knowledge embedded in its firmware or by having it explicitly communicated through an out-of-band channel once during system installation.

In most cases, static information describes the *format of on-disk structures*. For example, by knowing the format of the database log record, the semantic disk can observe each update operation to disk; by knowing the structure of B-Tree pages, the disk can determine which are internal pages versus leaf pages; finally, by understanding the format of data pages, the semantic disk can perform operations such as scanning the page to find "holes" when byte ranges are deleted. In other cases, static information describes the *location of on-disk structures*. For example, in Predator, knowing the names and IDs of *system catalog tables* such as the *RootIndex* and the _SINDXS table is useful.

3.2 Dynamic information

Dynamic information pertains to information about the DBMS that continually changes during operation. Examples of dynamic information include the particular set of disk blocks allocated to a certain table or whether a given disk block belongs to a table or to an index. Unlike static information, dynamic information needs to be continually tracked by the disk. To track dynamic information, a semantic disk utilizes static information about data structure formats to monitor changes to key data structures; these changes are then correlated to the higher level operations that could cause these changes.

Unfortunately, since both file systems and databases buffer and reorder writes, performing an accurate inference of higher level operations can be quite complex [28, 29]. To solve this problem, we use the technique of *log snooping*, in which the storage system observes the log records written out by the DBMS. With log snooping, the storage system leverages the fact that the database uses a write-ahead log (WAL) to track every operation that changes on-disk contents. Because of the WAL property, the log of an operation reaches disk *before* the effect of the operation; this strong ordering guarantee makes inferences underneath a DBMS accurate and straightforward.

Our implementation of log snooping is as follows. We assume that each log record contains a Log Sequence Number (LSN) [20]; the LSN is usually the byte offset of the start of that record in the log volume. The LSN allows the semantic disk to accurately infer the exact ordering of events that occurred in the database, even in the presence of group commits that can cause log blocks to arrive out of order. To order events, the disk maintains an *expected LSN* pointer, which is the LSN of the next log record expected to be seen by the disk; thus, when the semantic disk receives a write request to a log block, it knows exactly where in the block to look for the next log record. The semantic disk then processes that log record and advances the expected LSN pointer to point to the next record. Thus, even when log blocks arrive out of order, the semantic disk utilizes the LSN ordering to process the blocks in order; log blocks arriving out of order are *deferred* until the expected LSN reaches that block.

We now describe in more detail how our implementation of database-aware storage uses log snooping to infer four important pieces of dynamic information: transaction status, block ownership, block type, and relationships between blocks. We then describe the importance of a final piece of dynamic information: access statistics.

3.2.1 Transaction Status

A basic piece of dynamic information is the current state of each transaction that has been written to disk. Each transaction can be either *pending* or *committed* and a pending transaction may later be aborted. When performing work associated with a transaction, a semantic disk can choose to *pessimistically* recognize only committed transactions, or it can *optimistically* begin work on pending transactions as well. There are trade-offs to both the pessimistic and optimistic approaches.

The pessimistic approach is most appropriate when the semantic disk implements functionality that requires correctness. For example, when implementing secure delete

(Section 5), the semantic disk cannot shred data belonging to a pending transaction, given that the transaction may abort and the DBMS require the data again. However, the pessimistic approach will often have worse performance than the optimistic approach, since the pessimistic version must delay work and may require a significant amount of buffering. The optimistic approach is most beneficial when aborts are rare and the DBMS implements *group commits* (and may thus delay committing individual transactions for a long period).

Determining the status of each transaction is straightforward with log snooping. When the semantic disk observes that a new log record has been written, it adds it to a list of "pending" transactions; when the disk observes a `commit` record in the log, it determines which transactions have committed and moves them to a "committed" list.

3.2.2 Block Ownership

It is useful for a semantic disk to understand the logical grouping of blocks into tables and indices; this involves associating a block with the corresponding table or index *store* that logically *owns* the block. Performing this association in the semantic disk is relatively straight forward; since the effect of allocating a block must be recoverable, the DBMS first logs the operation before performing the allocation. Therefore, when the semantic disk later observes traffic to a disk block, it is simple to associate that block with the owning table or index. As we show later, in some cases it is sufficient for the semantic disk to map blocks to the store ID of the owning table, whereas in other cases, is useful for the semantic disk to further map the store ID to the actual table (or index) name.

For example, when allocating a block, SHORE writes out a `create_ext` log record with the block number and the ID of the owning store. When the semantic disk observes this log entry, it records the block number and store ID in an internal `block_to_store` hash table.

To further map the store ID to the actual table or index name, the disk uses static knowledge of the *system catalog tables*. In Predator, this mapping is maintained in a B-Tree called the *RootIndex*, whose logical store ID is statically known. Thus, when the disk observes `btree_add` records in the log with the *RootIndex* ID, the semantic disk is able to identify newly created mappings and add them to a `store_to_name` hash table.

3.2.3 Block Type

Another piece of useful information for a semantic disk is the *type* of a store (or a block); for example, whether a block is a data page or an index page. To track this information, the semantic disk again watches updates to the system catalog tables, the names of which are part of the static information known to the disk.

For example, in Predator, the _SINDXS table contains all indexes in the database; each tuple in _SINDXS contains the name of the index, the name of the table, and the attribute on which the index is built. The semantic disk detects inserts to this table by looking for the appropriate `page_insert` records in the log. The semantic disk is then able to determine whether a given block is part of a table or of an index by looking up its owning store in information derived from the _SINDXS table.

3.2.4 Block Relationships

A third type of useful information consists of the relationships across different blocks. One of the most useful relationships for a semantic disk to know is that between a table and the set of indices built on the table.

As stated above, in Predator, the association between indices and tables is kept the _SINDXS catalog table. Thus, a semantic disk can consult information derived from the _SINDXS table to associate a given table with its indices, or vice versa.

3.2.5 Access Patterns

In addition to the previous dynamic information, it is also useful for a semantic disk to know how tables and indexes are being accessed in the current workload. Although transaction status, block ownership, block type, and block relationships can be inferred relatively easily with log snooping, these access patterns are more difficult to infer.

Inferring access patterns was found to be relatively easy underneath a general-purpose file system [3, 30]. For example, the fact that a certain set of files lies within a directory implicitly conveys information to the storage system that those files are likely to be accessed together. Similarly, most file systems track the last time each file was accessed and periodically write this information to disk.

Although some modern database systems do track access statistics for performance diagnosis, the statistics are gathered at relatively coarse granularity; for example, the Automatic Workload Repository in Oracle 10g maintains access statistics [21].

Our experience has revealed that it would be useful for the DBMS to track three different types of statistics. Because this information is only used to optimize behavior, the DBMS can write the statistics periodically to disk (perhaps in additional catalog tables) without being transactional and thus can avoid the logging overhead.

The most basic statistic for the DBMS to communicate with the semantic disk is the *access time* of a particular block or table. This particular statistic is useful both in its own right and because it can be used to derive other statistics. A second useful statistic summarizes the *access correlation* between entities such as tables and indexes; for example, the DBMS could record for each query, the set of tables and indexes accessed. These correlation statistics capture the semantic groupings between different tables and is useful for collocating related tables. Finally, a third useful statistic tracks *access counts*, such as the number of queries that accessed a given table over a certain duration. This piece of information conveys the importance of various tables and indexes.

3.3 Case Studies

The actual static or dynamic information required within a database-aware disk depends upon the functionality that the disk is implementing. Therefore, we investigate a number of case studies that were previously implemented underneath of file systems. First, we investigate D-GRAID, a RAID system that degrades gracefully under failure [29]. Second, we implement a FADED, which guarantees that data is unrecoverable once the user deletes it [28]. Finally, we explore X-RAY, which implements a second-level storage-array cache [3].

4 Partial Availability with D-GRAID

Our first case study is to implement D-GRAID [29] underneath a DBMS. D-GRAID is a semantically-smart storage system that lays out blocks in a way that ensures graceful degradation of availability under unexpected multiple failures. Thus, D-GRAID enables continued operation of the system instead of complete unavailability under multiple failures. Previous work has shown that this approach significantly improves the availability of file systems [29].

In this section, we begin by reviewing the motivation for partial availability and D-GRAID. Next, we summarize our past experience when implementing D-GRAID underneath file systems. We then describe our techniques for implementing D-GRAID underneath a DBMS. Finally, we evaluate our version of D-GRAID and discuss its lessons.

4.1 Motivation

The importance of data availability cannot be over emphasized, especially in settings where downtime can cost millions of dollars per hour [18, 23]. To cope with failures, file systems and database systems store data in RAID arrays [22], which employ redundancy to automatically recover from a small number of disk failures.

Existing RAID schemes do not effectively handle catastrophic failures, that is, when the number of failures exceeds the tolerance threshold of the array (usually 1). Multiple failures occur due to two primary reasons. First, faults are often correlated [12]; a single controller fault or other component error can render a number of disks unavailable [7]. Second, system administration is the main source of failure in systems [11]. A large percentage of human failures occur during maintenance, where "the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure" (page 6) [11].

Under such extra failures, existing RAID schemes lead to complete unavailability of data until the contents of the array are restored from backup. This effect is especially severe in large arrays; for example, even if 30 out of 32 disks (roughly 94%) in a RAID-5 array are fully operational, the disk system (and consequently, the database) is completely unavailable.

This "availability cliff" arises because traditional storage systems employ simplistic layout techniques such as striping, that are oblivious of the semantic importance of blocks or relationships across blocks; when excess failures occur, the odds of semantically-meaningful data (*e.g.*, a table) remaining available are low. Furthermore, because modern storage arrays export abstract logical volumes which appear like a single disk [9, 34], the file system or DBMS has no control over data placement and cannot ensure that semantically-meaningful data remains after a single disk failure.

4.2 Filesystem-Aware D-GRAID

The basic goal of D-GRAID [29] is to make *semantically meaningful* fragments of data available under failures, so that workloads that access only those fragments can still run to completion, oblivious of data loss in other parts of the file system. By working on top of any redundancy technique (*e.g.*, RAID-1), D-GRAID provides graceful degradation when the number of failures exceed the tolerance threshold of the particular redundancy technique. When we implemented D-GRAID under a file system, we found three layout techniques to be important.

First, *fault-isolated data placement* is needed to ensure that semantic fragments remain available in their entirety. Under fault isolated placement, an entire semantic fragment is collocated within a single disk. We found that, for file system workloads, a reasonable semantic fragment consists of either a single file in its entirety (*i.e.*, its data blocks, its inode block, and potentially its indirect blocks) or all of the files in a single directory.

Second, *selective replication* is needed to ensure that essential meta-data and data that is always required is very likely to be available. In the file system context, this essential meta-data was found to consist of all directories (*i.e.*, data and inode blocks) and the structures of the file system (*i.e.*, superblock and bitmap blocks); the essential data was found to be the system binaries kept in known directories (*e.g.*, in `/usr/bin`, `/bin`, and `/lib`).

Third, *access-driven diffusion* in which popular data is striped across disks, is needed to improve throughput when a large file is placed on a single disk. We found that popular data could be dynamically identified by tracking logical segments without semantic knowledge; thus, access-driven diffusion can be implemented in the same manner whether beneath a file system or a DBMS.

4.3 Database-Aware D-GRAID

We now describe our techniques for implementing D-GRAID underneath a DBMS. First, we explore two techniques for fault-isolated data placement that target widely different database usage patterns: moderately-sized tables which can use coarse-grained fragmentation and large tables which must use fine-grained fragmentation. Second, we explore the structures that need to be selectively replicated. Third, we describe our implementation of access-driven diffusion. Finally, we describe *infallible writes*, a new technique that was not required for file systems.

When identifying semantic fragments, there are two fundamental differences under a DBMS versus a file system. First, in a DBMS, one is more likely to find extremely large tables that will not fit within a single disk. Therefore, we describe our techniques separately for moderately-sized tables, which can use coarse-grained fragmentation and fit an entire table within a disk, and for very large tables, which must use fine-grained fragmentation and stripe tables and indexes across multiple disks. Second, in a DBMS, the queries being performed directly impact which tables and indexes are accessed together. Therefore, we describe how semantic groupings are affected by three popular types of queries: scans, index lookups, and joins.

4.3.1 Fault-Isolated Placement: Coarse-Grained

The simplest case occurs when the database contains a large number of moderately-sized tables; in this situation, a semantic fragment can be defined in terms of an entire table. We now present layout strategies for improved availability for each query type given this scenario.

A. Scans: Many queries, such as selection queries that filter on a non-indexed attribute or aggregate queries on a single table, involve a sequential scan of one entire table. Since a scan requires the entire table to be available in order to succeed, a simple choice of a semantic fragment is the set of all blocks belonging to a table; thus, an entire table is placed within a single disk, so that when failures occur, a subset of tables are still available in their entirety, and therefore scans just involving those tables will continue to operate oblivious of failure.

B. Index lookups: Index lookups form another common class of queries. When a selection condition is applied based on an indexed attribute, the DBMS looks up the corresponding index to find the appropriate tuple record IDs, and then reads the relevant data pages to retrieve the tuples. Since traversing the index requires access to multiple pages in the index, collocation of a whole index improves availability. However, if the index and table are viewed independently for placement, an index query fails if either the index or the table is unavailable, decreasing availability. Thus, a better strategy to improve availability is to collocate a table with its indexes. We call the latter strategy *dependent index placement*.

C. Joins: Many queries involve joins of multiple tables. Such queries typically require all the joined tables to be available, in order to succeed. To improve availability of join queries, D-GRAID collocates tables that are likely to be joined together into a single semantic fragment, which is then laid out on a single disk. Identification of such "join groups" requires extra access statistics to be tracked by the DBMS.

For our implementation, we modified the Predator DBMS to record the set of stores (tables and indexes) accessed for each query and to construct a matrix that indicates the access correlation between each pair of stores. This information is written to disk periodically (once every 5 seconds). These modifications to Predator are relatively straight-forward, involving less than 200 lines of code. D-GRAID then uses this information to collocate tables that are likely to be accessed together.

4.3.2 Fault-Isolated Placement: Fine-Grained

While collocation of entire tables and indexes within a single disk provides enhanced availability, a single table or index may be too large to fit within a single disk, even though disk capacities are roughly doubling every year [13]. In such a scenario, we require a fine-grained approach to semantic fragmentation. In this approach, D-GRAID stripes tables and indexes across multiple disks (similar to a traditional RAID array), but adopts new techniques to enable graceful degradation, as detailed below.

A. Scans: Scans fundamentally require the entire table to be available, and thus any striping strategy will impact availability of scan queries. To help availability, a hierarchical approach is possible: a large table can be split across the minimal number of disks that can hold it, and the disk group can be treated as a logical fault-boundary; D-GRAID can be applied over such logical fault-boundaries. Alternatively, if the database supports approximate queries [15], it can provide partial availability for scan queries even with missing data.

B. Index lookups: With large tables, index-based queries are likely to be more common. For example, an OLTP workload such as TPC-C normally involves index lookups on a small number of large tables. These queries do not require the entire index or table to be available. D-GRAID uses two simple techniques to improve availability for such queries. First, the internal pages of the B-tree index are aggressively replicated, so that a failure does not take away, for instance, the root of the B-tree. Second, an index page is collocated with the data pages corresponding to the tuples pointed to by the index page. For this collocation, D-GRAID uses a probabilistic strategy; when a leaf index page is written, D-GRAID examines the set of RIDs contained in the page, and for each RID, determines which disk the corresponding tuple is placed in. It then places the index page on the disk which has the greatest number of matching tuples. Note that we assume the table is clustered on the index attribute; page-level collocation may not be effective in the case of non-clustered indexes.

C. Joins: Similar to indexes, page-level collocation can also be applied across tables of a join group. For such collocation to be feasible, all tables in the join group should be clustered on their join attribute. Alternatively, if some tables in the join group are "small", they can be replicated across disks where the larger tables are striped.

4.3.3 Selective Replication

There are some data structures within a DBMS that must be available for *any* query in the system to be able to run. For example, system catalogs (that contain information about each table and index) are frequently consulted; if such structures are unavailable under partial failure, the fact that most data remains accessible is of no practical use. Therefore, D-GRAID aggressively replicates the system catalogs and the *extent map* in the database that tracks allocation of blocks to stores. In our experiments, we employ 8-way replication of important meta-data; we believe that 8-way replication is quite feasible given the "read-mostly" nature of such meta-data and the minimal space overhead (less than 1%) this entails.

The database log plays a salient role in the recoverability of the database, and its ability to make use of partial availability. It is therefore important for the log to be available under multiple failures. We believe that providing high availability for the log is indeed possible. Given that the size of the "active portion" of the log is determined by the length of the longest transaction factored by the concurrency in the workload, the portion of the log that needs to be kept highly available is quite reasonable. Modern storage arrays have large amounts of persistent RAM, which are obvious locations to place the log for high availability, perhaps replicating it across multiple NVRAM stores. This, in addition to normal on-disk storage of the log, can ensure that the log remains accessible in the face of multiple disk failures.

4.3.4 Access-Driven Diffusion

As stated above, with coarse-grained fragmentation, an entire table is placed within a single disk. If the table is large or is accessed frequently, this can have a performance impact since the parallelism that can be obtained across the disks is wasted. To remedy this, D-GRAID monitors accesses to the logical address space and tracks logical segments that are likely to benefit from parallelism. D-GRAID then creates an extra copy of those blocks and spreads them across the disks in the array, like a normal RAID would do. Thus, for blocks that are "hot", D-GRAID regains the lost parallelism due to collocated layout, while still providing partial availability guarantees. Reads and writes are first sent to the diffused copy, with background updates being sent to the actual copy. This technique underneath of a DBMS is essentially identical to that used underneath a file system.

4.3.5 Infallible Writes

Partial availability of data introduces interesting problems for the transaction and recovery mechanisms within a DBMS. For example, a transaction is often declared "committed" after it is reflected in the log. In a partially available system, after a crash, a redo for the transaction can fail if some pages are not available, which may seem to affect the durability semantics of transactions. However, this problem has already been considered and solved in ARIES [20], in the context of handling offline objects during *deferred restart*.

To ensure transaction durability, D-GRAID implements infallible writes, in which it guarantees that a write "always" succeeds. If a block to be written is destined for a dead disk, D-GRAID remaps it into a live disk and writes it (assuming that there is free space remaining on a live disk). This remapping prevents a new failure when flushing an already committed transaction to disk.

4.4 Evaluation

We evaluate the availability improvements and performance of D-GRAID through a prototype implementation; our D-GRAID prototype functions as a software RAID driver in the Linux 2.4 kernel, and operates underneath the Predator/Shore DBMS.

4.4.1 Availability Improvements

To evaluate availability improvements with D-GRAID, we use a D-GRAID array of 16 disks, and study the fraction of queries that the database serves successfully under an increasing number of disk failures. Since layout techniques in D-GRAID are complementary to existing RAID schemes such as parity or mirroring, we show D-GRAID Level 0 (*i.e.*, no redundancy for data) in our measurements, for simplicity. We mainly use microbenchmarks to analyze the availability provided by various layout techniques in D-GRAID.

A. Coarse-grained fragmentation

We first evaluate the availability improvements due to the coarse-grained fragmentation techniques in D-GRAID. Figure 1 presents the availability of scan, index lookup, and join queries for synthetic workloads under multiple disk failures. The percentage of such queries that complete successfully is reported.

The leftmost graph in Figure 1 shows the availability for scan queries. The database had 200 tables, each with 10,000 tuples. The workload is as follows: each query chooses a table at random and computes an average over a non-indexed attribute, thus requiring a scan of the entire table. As the graph shows, collocation of whole tables enables the database to be partially available, serving a proportional fraction of queries. In comparison, just one

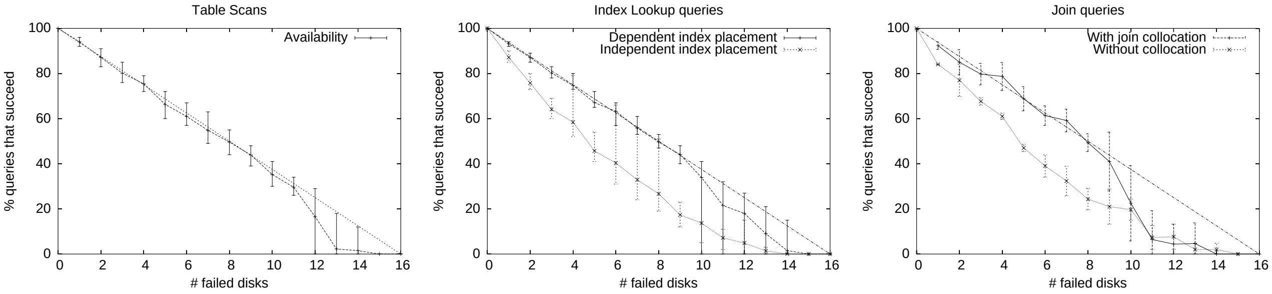

Figure 1: **Coarse-grained fragmentation.** *The graphs show the availability degradation for scans, index lookups and joins under varying number of disk failures. A 16-disk D-GRAID array was used. The steeper fall in availability for higher number of failures is due to the limited (8-way) replication of metadata. The straight diagonal line depicts "ideal" linear degradation.*

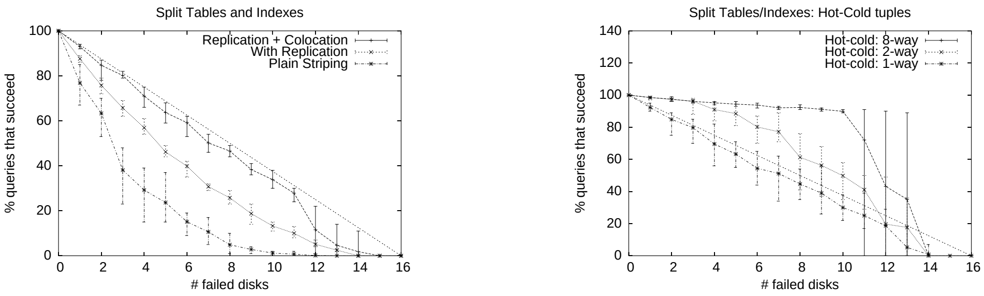

Figure 2: **Index Lookups under fine-grained fragmentation.** *The graphs show the availability degradation for index lookup queries. The left graph considers a uniformly random workload, while the right graph considers a workload where a small set of tupes are very popular*

failure in a traditional RAID-0 system results in complete unavailability. Note that if redundancy is maintained (*i.e.*, parity or mirroring), both D-GRAID and traditional RAID will tolerate up to one failure without any availability loss.

The middle graph in Figure 1 shows the availability for index lookup queries under a similar workload. We consider two different layouts; in both layouts, an entire "store" (*i.e.*, an index or a table) is collocated within one disk. In *independent index placement*, D-GRAID treats the index and table as independent stores and hence possibly allocates different disks for them, while with *dependent index placement*, D-GRAID carefully allocates the index on the same disk as the corresponding table. As can be seen, dependent placement leads to much better availability under failure.

Finally, to evaluate the benefits of join-group collocation, we use the following micro-benchmark: the database contains 100 pairs of tables, with joins always involving tables in the same pair. We then have join queries randomly select a pair and join the corresponding two tables. The rightmost graph in Figure 1 shows that by collocating joined tables, D-GRAID achieves higher availability.

B. Fine-grained fragmentation

We now evaluate the effectiveness of fine-grained fragmentation. We focus on the availability of index lookup queries since they are the most interesting in this category. The workload we use for this study consists of index lookup queries on randomly chosen values of a primary key attribute in a single large table. We plot the fraction of queries that succeed under varying number of disk failures. The left graph in Figure 2 shows the results.

There are three layouts examined in this graph. The lowermost line shows availability under simple striping with just replication of system catalogs. As can be seen, the availability falls drastically under multiple failures due to loss of internal B-tree nodes. The middle line depicts the case where internal B-tree nodes are replicated aggressively; as can be expected, this achieves better availability. Finally, the third line shows the availability when data and index pages are collocated, in addition to internal B-tree replication. Together, these two techniques ensure near linear degradation of availability.

The right graph in Figure 2 considers a similar workload, but a small subset of tuples are much "hotter" compared to the others. Specifically, 5% of the tuples are accessed in 90% of the queries. Even under such a workload, simple replication and collocation provide near linear degradation in availability since hot pages are spread nearly uniformly across the disks. However, under such a hot-cold workload, D-GRAID can improve availability further by replicating data and index pages containing

	D-GRAID	RAID-0	Slowdown
Table Scan	7.97 s	6.74 s	18.1%
Index Lookup	51 ms	49.7 ms	2.7%
Bulk Load	186.61 s	176.14 s	5.9%
Table Insert	11.4 ms	11 ms	3.6%

Table 1: **Time Overheads of D-GRAID.** *The table compares the performance of D-GRAID under fine-grained fragmentation, with default RAID-0 under various microbenchmarks. An array of 4 disks is used.*

	Scan Time (s)
RAID-0	6.74
D-GRAID	15.69
D-GRAID + Diffusion	7.35

Table 2: **Diffusing Collocated Tables.** *The table shows the scan performance on a 4-disk array under coarse-grained fragmentation.*

such hot tuples. The other two lines depict availability when such hot pages are replicated by factors of 2 and 8. Thus, when a small fraction of (read mostly) data is hot, D-GRAID utilizes that information to enhance availability through selective replication.

4.4.2 Performance overheads

We now evaluate the performance implications of fault-isolated layout in D-GRAID. For all experiments in this section, we use a 4-disk D-GRAID array comprised of 9.1 GB IBM UltraStar 9LZX disks with peak throughput of 20 MB/s. The database used has a single table of 500,000 records, each sized 110 bytes, with an index on the primary key.

A. Time and space overheads

We first explore the time and space overheads incurred by our D-GRAID prototype for tracking information about the database and laying out blocks to facilitate graceful degradation. Table 1 compares the performance of D-GRAID with fine-grained fragmentation to Linux software RAID 0 under various basic query workloads. The workloads examined are a scan of the entire table, an index lookup of a random key in the table, bulk load of the entire indexed table, and inserts into the indexed table. D-GRAID performs within 6% of RAID-0 for all workloads except scans. The poor performance in scans is due to a Predator anomaly, where the scan workload completely saturated the CPU (6.74 s for a 50 MB table across 4 disks). Thus, the extra CPU cycles required by D-GRAID impacts the scan performance by about 18%. This interference is because our prototype competes for resources with the host; in a hardware RAID system, such interference would not exist. Overall, we find that the overheads of D-GRAID are quite reasonable.

We also evaluated the space overheads due to aggressive metadata replication and found them to be minimal; the overhead scales with the number of tables, and even in a database with 10,000 tables, the overhead is only about 0.9% for 8-way replication of important data.

B. Access-driven Diffusion

We now evaluate the benefits of diffusing an extra copy of popular tables. Table 2 shows the time taken for a scan of the table described above, under coarse-grained fragmentation in D-GRAID. As can be seen, simple collocation leads to poor scan performance due to the lost parallelism. With the extra diffusion aimed at performance, D-GRAID performs much closer to default RAID-0.

4.5 Comparison

In our implementation of D-GRAID underneath a DBMS, we uncovered some fundamental challenges that were not present under a file system. First, the notion of semantically-related groups is more complex in a DBMS because of the various inter-relationships that exist across tables and indexes. In the file system case, whole files or whole directories were reasonable approximations of semantic groupings. In a DBMS, since the goal of D-GRAID is to enable serving as many higher level queries as possible, the notion of semantic grouping is *dynamic*, *i.e.*, it depends on the query workload. Second, identifying "popular" data that needs to be aggressively replicated, is easier in file systems; standard system binaries and libraries were obvious targets, independent of the specific file system running above. However, in a DBMS, the set of popular tables varies with the DBMS and is often dependent on the query workload. Thus, effectively implementing D-GRAID underneath a DBMS requires slightly modifying the DBMS to record additional information. Finally, to ensure transaction durability, we implemented infallible writes for the version under the DBMS.

Comparing how well D-GRAID performs beneath a DBMS versus a file system we see many similarities. For example, both versions of D-GRAID successfully enable graceful degradation of availability; that is, both versions enable at least the expected number of processes or queries to complete successfully, given a fixed number of disk failures. In fact, both versions enable *more* than the expected number to complete when a subset of the data is especially popular. Similarly, both versions of D-GRAID do introduce some time overhead; interestingly, the slowdowns for our database version are generally lower than those for the file system version. Finally, both versions require access-driven diffusion to obtain acceptable performance.

5 Secure Delete with FADED

Our second case study is to implement FADED [28] underneath a DBMS. FADED is a semantically smart disk that detects deletes of records and tables at the DBMS level and securely overwrites (*i.e.*, *shreds*) the relevant data to make it irrecoverable. We extend previous work that implemented the same functionality for file systems [28].

5.1 Motivation

Deleting data such that recovery is impossible is important for file system security [4, 14]. Government regulations require guarantees on sensitive data being *forgotten*, and such requirements could become more important in databases [2]. Recent legislations on data retention, such as the Sarbanes-Oxley Act, have accentuated the importance of secure deletion.

Secure deletion of data in magnetic disks involves overwriting disk blocks with a sequence of writes with certain specific patterns to cancel out remnant magnetic effects due to past layers of data in the block. While early work indicated that as many as 32 overwrites per block are required for secure erase [14], recent work shows that two to three such overwrites suffice for modern disks [16].

Neither a file system nor a DBMS can ensure secure deletion when it functions on top of modern storage systems, which transparently perform various optimizations. For example, the storage system could buffer writes in NVRAM before writing them out to disk [34]. In the presence of NVRAM buffering, multiple overwrites done by the file system or DBMS may be collapsed into a single write to the physical disk, making the overwrites ineffective. Also, in the presence of block migration within the storage system [9], overwrites by the file system or DBMS will miss past copies.

Thus, secure deletion requires the low level information and control that the storage system has, and at the same time, higher level semantic information about the file system or DBMS to detect logical deletes. A semantically-smart disk system is thus an ideal locale to implement secure deletion.

5.2 Filesystem-Aware FADED

When running underneath a file system, FADED infers that a file is deleted by tracking writes to inodes, indirect blocks, and bitmap blocks and looking for changes. Due to the asynchronous nature of file systems, FADED is not able to guarantee that the current contents of a block belong to the deleted file and not to a newly allocated file (which should not be shredded). To ensure that it does not shred valid data, FADED uses *conservative overwrites* in which it shreds only an old version of a block before restoring the current contents of the block.

In previous work, we implemented FADED for three file systems: Linux ext2, Linux ext3, and Windows VFAT. However, for FADED to work correctly, each file system had to be changed. For example, Linux ext2 was modified to ensure that data bitmap blocks are flushed whenever an indirect block is allocated or freed; Windows VFAT was changed to track a generation number for each file; finally, Linux ext3 was modified so that the list of modified data blocks are included in each transaction.

5.3 Database-Aware FADED

To implement FADED beneath a DBMS, the semantic disk must be able to identify and handle deletes for both entire tables as well as for individual records. We discuss these two cases in turn.

The simplest case for FADED is when a whole table is deleted. When a `drop table` command is issued, FADED must shred all blocks belonging to the table. FADED uses log snooping to identify log records that indicate freeing of extents from stores. In SHORE, a `free_ext_list` log record is written for every extent freed. Once FADED knows the list of freed blocks, it can issue secure overwrites to those pages. If the transaction aborts (thus undoing the deletes), the contents of the freed pages will be required; therefore, FADED pessimistically waits until the transaction is committed before performing any overwrites.

Handling record-level deletes in FADED is more challenging. When specific tuples are deleted (via the SQL `delete from` statement), specific byte ranges in the pages containing those tuples must be shredded. On a delete, a DBMS typically marks the relevant page "slot" free, and increments the free space count in the page. Since such freeing of slots is logged, FADED can learn of such record deletes by log snooping. However, FADED cannot shred the whole page because other records in the page could still be valid. Rather than read the current page from disk, we defer the shredding until FADED receives a write to the page reflecting the relevant delete. On receiving such a write, FADED shreds the entire page in the disk, and then writes the new data received. However, there are two complications with this basic technique.

The first complication is to identify the correct version of the page containing the deleted record. Assume that FADED observes a record delete d in page P, and waits for a subsequent write of P. When P is written, FADED needs to detect if the version written reflects d. The version could be stale if the DBMS wrote the page sometime before the delete, but the block was reordered by the disk scheduler and arrives later at the disk. This issue is similar to that of the file-system version of FADED; however, rather than use conservative overwrites, the database-aware version uses the WAL property of the DBMS to ensure correct operation. Specifically, database-aware FADED uses the *PageLSN* field in the page [20] to identify whether P reflects the delete. The *PageLSN* of a page tracks the sequence number of the

	Run time (s)	
	Workload I	Workload II
Default	52.0	66.0
$FADED_2$	78.3	128.5
$FADED_4$	91.0	160.0
$FADED_6$	104.5	190.2

Table 3: **Overheads of secure deletion.** *This table shows the performance of FADED with 2,4 and 6 overwrites, under two workloads. Workload I deletes contiguous records, while Workload II deletes records randomly across the table.*

latest log record describing a change in the page. Thus, FADED simply needs to compare the *PageLSN* to the LSN of the delete *d*.

The second complication is that the DBMS may not zero out bytes that belonged to deleted records; as a result, old data still remains in the page. Thus, when FADED observes the page write, it scans the page looking for free space and explicitly zeroes out the deleted byte ranges. Since the page could remain in the DBMS cache, all subsequent writes to the page must also be scanned and zeroed out appropriately.

5.4 Evaluation

We now briefly evaluate the cost of secure deletion in FADED through a prototype implementation. The prototype is implemented as a device driver in the Linux 2.4 kernel, and works underneath Predator [27].

We consider two workloads operating on a table with 500,000 110-byte records. In the first workload, we perform a `delete from` in such a way that all rows in the second half of the table are deleted (*i.e.*, the deleted pages are contiguous). In the second workload, the tuples to be deleted are selected in random.

Table 3 compares the default case without FADED to FADED using two, four, and six overwrite passes. As expected, secure deletion comes at a performance cost due to the extra disk I/O for the multiple passes of overwrites. Given that modern disks can effectively shred data with only two overwrites [16], we focus on $FADED_2$; in this case, performance is 50% to 95% slower. However, since such overhead is incurred only on deletes, and only sensitive data needs to be deleted in this manner, we believe the costs are reasonable in situations where the additional security is required.

5.5 Comparison

The primary difference between the two versions of FADED is that the database-aware version is able to leverage the transactional properties of the DBMS to definitively track whether a particular block should be shredded. As a result, while the file system version of FADED required changes to the file system (with the exception of data journaled ext3), our implementation of FADED does not require any DBMS changes. However, our version does require detailed information about the on-disk page layout of the DBMS. Furthermore, the record-level granularity of deletes in a DBMS makes secure deletion more complex than in its file system counterpart.

Both versions of FADED incur some overhead, depending upon the workload and the number of overwrites. On our two delete-intensive database workloads, FADED was 50% or 95% slower with two overwrites. Similarly, for the two file system workloads, FADED was 51% to 280% slower with two overwrites (from Table 11 of [28]). In summary, the slowdown incurred by FADED depends more on the workload and the number of overwrites than on whether it is used by a DBMS or a file system.

6 Exclusive Caching with X-RAY

Our final case study is to implement X-RAY [3] underneath a DBMS. X-RAY is an exclusive caching mechanism for storage arrays that attempts to cache disk blocks which are *not* present in the higher-level buffer cache, thus providing the illusion of a single large LRU cache. Previous work has demonstrated that this approach performs very well when the buffer cache is maintained by a file system [3].

6.1 Motivation

Modern storage arrays possess large amounts of RAM for caching disk blocks. For instance, a high-end EMC storage array has up to 256 GB of main memory for caching. Typically, this cache is a second-level cache and the file system or a database system maintains its own buffer cache in the host main memory. Current caching mechanisms in storage arrays do not account for this; a block is placed in the array cache on a read, duplicating the same blocks cached above. Cache space is thus wasted due to inclusion. A better strategy would be for the contents of the buffer cache and the disk array cache to be *exclusive*.

Wong et al. [35] proposed to avoid cache inclusion by modifying the file system and the disk interface to support a SCSI "demote" command, which enables treating the disk array cache as a victim cache. For a database system, their approach would require the DBMS to inform the disk about evictions from its buffer pool. However, requiring an explicit change to the SCSI storage interface makes this scheme hard to deploy, since industry consensus is required for adopting such a change.

6.2 Filesystem-Aware X-RAY

X-RAY predicts the contents of the file system buffer pool and then chooses to cache only the most recent victims in its own cache; X-RAY requires no changes to the storage interface. X-RAY uses access time statistics (*i.e.*, which block was accessed and when) to perform its predictions;

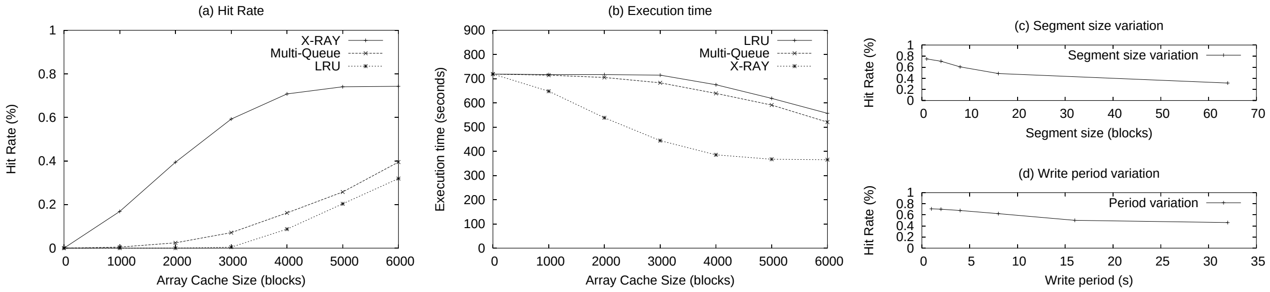

Figure 3: **X-RAY Performance.** *The figure presents an evaluation of X-RAY under the TPC-C benchmark. The DBMS buffer cache was set to 6000 blocks for these studies. (a) The hit rate of X-RAY is compared to other caching mechanisms. The segment size is 4 blocks, and access information is written every 1 second. (b) The corresponding execution times are compared. The times are based on a buffer cache hit time of 20 μs, a disk array cache hit time of 0.2 ms, a disk read time of 3 ms to 10 ms. (c) Hit-rate of X-RAY is measured for different segment sizes; the write period is kept at 1 second. (d) The write period is varied and the X-RAY hit rate is measured. The segment size is kept at 4 blocks.*

for file systems such as Linux ext2, access statistics are recorded at the granularity of a file and are directly available in file inodes.

X-RAY uses these access statistics to maintain an ordered list of block numbers, from the LRU block to the MRU block. This is complicated by the fact that access statistics are tracked only a per-file basis. The ordered list is updated when X-RAY obtains new information, such as when the file system reads from disk (making the read block the most recently accessed) and when the file system writes an access time to disk. When a disk read arrives to a block A, X-RAY infers that A was evicted from the buffer cache some time in the past; it can also infer that any block B with an earlier access time was evicted as well (assuming an LRU policy). If the access time of block A is updated, but X-RAY did not observe a disk read for A, then X-RAY infers that block A, and all blocks with a later access time than A, are present in the buffer cache.

If the higher-level cache policy is LRU (which is the usual case), blocks close to the MRU end of the list are predicted to be in the file system buffer cache. The blocks near the LRU part of the list are considered the exclusive set; X-RAY caches the most recent blocks in the exclusive set, using extra internal array bandwidth or idle time between disk requests to read these blocks into the cache.

6.3 Database-Aware X-RAY

The database-aware version of X-RAY is very similar to the file system-aware version. The primary difference in creating a database-aware X-RAY occurs because a DBMS does not typically track access statistics. Although some database systems do maintain access statistics for administrative purposes (*e.g.*, AWR [21] for Oracle), these statistics are coarse in granularity and are written out only after long intervals.

Therefore, to implement database-aware X-RAY we must modify the database buffer manager to write out access statistics periodically. Specifically, each table or index is divided into fixed-sized *segments*, and the buffer manager periodically writes to disk the access time for segments accessed during the last period of time. X-RAY assumes that all blocks in the segment have been accessed when it sees that the access time statistic is updated. Thus, the accuracy with which X-RAY can predict the contents of the database cache is sensitive to both the size of each segment and the update interval. One advantage of explicitly adding this information is that we can tune the implementation by changing the size of the segment or the update interval. An alternative to adding this access information would be to modify the DBMS to directly report when it has evicted a block from its own cache, as in DEMOTE [35]. We believe that adding just access statistics is a better approach because the statistics are more general and can be used by semantic disks implementing other functionality (*e.g.*, D-GRAID [29]).

6.4 Evaluation

We evaluate the performance of our database-aware version of X-RAY with a simulation of both the database buffer cache and the disk array cache; the evaluation of the filesystem-aware X-RAY was performed using a simulation as well. The database buffer cache is maintained in LRU fashion; the DBMS periodically writes out access information at the granularity of one segment. The array cache is managed by X-RAY. We assume that X-RAY has sufficient internal bandwidth for its block reads.

We instrumented the buffer cache manager of the Postgres DBMS [24] to generate traces of page requests at the buffer cache level. We use Postgres because Predator does not have a programming API in Linux, which is required to implement TPC-C. We use an approximate implementation of the TPC-C benchmark for our evaluation (it adheres to the TPC-C specification [32] in its access pattern). A total of 5200 transactions are performed.

We evaluate the performance of X-RAY in terms of array cache hit rate and execution time. We compare X-

	Static				Dynamic				
	Catalog tables	Log record format	B-tree page format	Data page format	Transaction status	Block ownership	Block type	Block relationships	Access statistics
D-GRAID									
basic	×	×				×	×	×	
+fine-grained frags	×	×	×			×	×	×	
+join-collocation	×	×				×	×	×	×
FADED									
basic	×	×			×	×	×		
+record-level delete	×	×		×	×	×	×		
X-RAY									
basic	×	×				×			×

Table 4: **DBMS Information required for case studies.** *The table lists the static information that must be embedded into the semantic disk and the dynamic state that is automatically tracked by the disk.*

RAY to plain LRU and the Multi-Queue mechanism [36] designed for second level caches. We also explore sensitivity to segment size and access time update periodicity.

Figure 3a compares the hit rate of X-RAY with that of the other schemes and Figure 3b compares the corresponding execution times. The segment size is set to four blocks and access information is written out every second for this study. We see that X-RAY has much better hit rate than both LRU and Multi-Queue. This hit rate advantage extends to execution time despite the overhead of writing out the access information; X-RAY performs up to 75% better than LRU and up to 65% better than Multi-Queue.

Figure 3c evaluates the sensitivity of the X-RAY cache hit rate to segment size. As expected, the hit rate drops slightly with an increase in segment size. Figure 3d shows sensitivity to the access information update interval. We see that X-RAY can tolerate a reasonable delay (*e.g.*, about 5 seconds) when obtaining access updates.

6.5 Comparison

The file system and database versions of X-RAY are quite similar. To implement X-RAY, the semantic disk requires access statistics; that is, it must know which blocks are being accessed by the layer above. Although most file systems track and periodically write such statistics, a DBMS does not. Therefore, to use X-RAY, the DBMS must be modified to explicitly track access times for segments within each table. One advantage of explicitly adding this information is that one can tune the statistics more appropriately (*i.e.*, the size of segment and the update interval). Whether running beneath a file system or a database, X-RAY was found to substantially improve the array cache hit rate, relative to both LRU and Multi-Queue.

7 Information for Case Studies

In this section, we review the static and dynamic information required within a database-aware disk, given that this needed information depends upon the functionality that it is being implemented. The exact information required for variants of our three case studies is listed in Table 4.

Probably the biggest concern for database vendors is the static information that must be exported; for example, if a storage system understands the format of a particular catalog table, then the database vendor may be loathe to change its format. The amount of static information varies quite a bit across the case studies. While all of our case studies must know the format of catalog tables and log records, only D-GRAID with support for fine-grained fragmentation and FADED with record-level deletes need more detailed knowledge, such as the B-tree page format and the data page format, respectively.

The useful dynamic information also varies across case studies. The most fundamental piece of dynamic information is block ownership, as shown by the fact that it is required by every case study; block type is also a generally useful property, needed by both D-GRAID and FADED. The other pieces of dynamic information are not widespread. For example, only FADED needs to know precisely when a transaction has committed, since to be correct, it must be pessimistic in determining when to overwrite data; only D-GRAID needs to be able to associate blocks from a table with the blocks from the corresponding index, and vice versa. Finally, access correlation and access count statistics are needed by one of the D-GRAID variants to collocate related tables and to aggressively replicate "hot" data; the simple access time statistic is needed by X-RAY to predict the contents of the higher-level buffer cache.

8 Conclusions

> "Today we [the database community] have this sort of simple-minded model that a disk is one arm on one platter and [it holds the whole database]. And in fact [what's holding the database] is RAID arrays, it's storage area networks, it's all kinds of different architectures underneath that hood, and it's all masked over by a logical volume manager written by operating system people who may or may not know anything about databases. Some of that transparency is really good because it makes us more productive and they just take care of the details. ... But on the other hand, optimizing the entire stack would be even better. So, we [in the two fields] need to talk, but on the other hand we want to accept some of the things that they're willing to do for us." [26].
> *-Pat Selinger*

Semantic knowledge in the storage system enables powerful new functionality to be constructed. For example, the storage system can improve performance with better caching [3], can improve reliability [29], and can provide additional security guarantees [28]. In this paper, we have shown that semantic storage technology can be deployed not only beneath commodity file systems, but beneath database management systems as well.

We have found that some different techniques are required to handle database systems. First, we investigated the impact of transactional semantics within the DBMS. In most cases, transactions simplify the work of a semantic disk. For example, log snooping enables the storage system to observe the operations performed by the DBMS and to definitively infer dynamic information without changing the DBMS. However, the storage system must also ensure that it does not interfere with the transactional semantics. For example, we found that infallible writes are useful to ensure transaction durability after some disks have failed. Second, we explored how the lack of access statistics within a DBMS complicates its interactions with a semantic disk. In this case, we found that it was helpful to slightly modify the database system to gather and relay simple statistics.

Acknowledgements

We would like to thank David Black for encouraging us to extend the semantic disks work to databases. We also thank David DeWitt, Jeff Naughton, Rajasekar Krishnamurthy and Vijayan Prabhakaran for their insightful comments on earlier drafts of this paper, and Jeniffer Beckham for pointing to us the Pat Selinger quote. Finally, we thank the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved this paper.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0325267, IBM, Network Appliance, and EMC.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th VLDB*, 2002.

[3] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04*, 2004.

[4] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *USENIX Security*, August 2001.

[5] H. Boral and D. J. DeWitt. Database Machines: An Idea Whose Time has Passed? In *3rd International Workshop on Database Machines*, 1983.

[6] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.

[7] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, pages 177–190, 2002.

[9] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2005.

[10] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[11] J. Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[12] S. D. Gribble. Robustness in Complex Systems. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[13] E. Grochowski. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. *Datatech*, September 1999.

[14] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security*, July 1996.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD '97*, pages 171–182, 1997.

[16] G. Hughes and T. Coughlin. Secure Erase of Disk Drive Data. IDEMA Insight Magazine, 2002.

[17] K. Keeton. *Computer Architecture Support for Database Applications*. PhD thesis, University of California at Berkeley, 1999.

[18] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[19] M. Carey et. al. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD Conference*, 1994.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, March 1992.

[21] Oracle. The Self-managing Database: Automatic Performance Diagnosis. https://www.oracleworld2003.com/published/40092/40092.doc, 2003.

[22] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, 1988.

[23] D. A. Patterson. Availability and Maintainability $>>$ Performance: New Focus for a New Century. Key Note Lecture at FAST '02, 2002.

[24] Postgres. The PostgreSQL Database. http://www.postgresql.com.

[25] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *VLDB 24*, 1998.

[26] P. Selinger and M. Winslett. Pat Selinger Speaks Out. *SIGMOD Record*, 32(4):93–103, December 2003.

[27] P. Seshadri and M. Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. In *SIGMOD '97*, 1997.

[28] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[29] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, 2004.

[30] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, 2003.

[31] D. Slotnick. *Logic Per Track Devices*, volume 10, pages 291–296. Academic Press, 1970.

[32] TPC-C. Transaction Processing Performance Council. http://www.tpc.org/tpcc/.

[33] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[34] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[35] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX '02*, 2002.

[36] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX '01*, pages 91–104, 2001.