Database Integrity: Security, Reliability, and Performance Considerations

Susan Gordon
Indiana University South Bend
1700 Mishawaka Avenue, South Bend, IN 46634
slgordon@iusb.edu

Abstract

Database integrity is a central underlying issue in the implementation of database technology. Trust in the correctness of the data that is held by the database system is a prerequisite for using the data in business, research or decision making applications. This paper will begin by discussing the areas that pose challenges in ensuring database security and reliability. It will look at the benefits and limitations of possible hardware and software solution strategies, especially with respect to the considerations of system overhead and the effect on system performance. It will specifically discuss the use of error checking and correction codes to address integrity issues and consider how these codes may be used to help improve the performance of database systems.

1. Introduction

In a database system, a method to ensure data integrity is fundamental to providing database reliability and security. In particular, as data is communicated or distributed over networks, a method to validate information as authentic is required. The value of a database is dependent upon a user's ability to trust the completeness and soundness of the information contained in the data [1].

Integrity requires that data is protected from improper modification, and integrity is lost if unauthorized changes are made by intent or by accident [1]. Database integrity problems can have many sources. A problem may be caused by a hardware malfunction, a software bug, an attack on a system, or a user error. The undesirable changes to a database may also be classified broadly as malicious and non-malicious [2], in other words, keeping unauthorized users from accessing or changing the data and keeping authorized users from accidentally corrupting the data.

There are many strategies to try to avoid, detect, and correct problems. Avoidance strategies include encrypting data, journaling, or using read-only storage in appropriate situations. Errors may be detected by replicating or mirroring data, parity checking, and the use of checksums generated by several kinds of hash functions. Correction may be accomplished by majority vote if mirroring is done with more than one copy, by the use of RAID level 5 disks, or by applying error correction codes such as Hamming codes.

If loss of integrity is not corrected, the continued use of corrupted data could result in further damage, inaccuracy, or erroneous decisions [1].

Some methods used to ensure database integrity may also have some additional welcome side-effects. They may be able to enhance database security by detecting unauthorized modifications to files. They may enhance database performance if a system can take advantage of the redundant information required. Integrity checking may also be able to identify a hardware failure in a disk by detecting data corruption [1].

This paper begins by looking at storage system problems and solutions and available database tools and strategies. It will then specifically consider the role of error correcting codes and secure hash algorithms in providing data security and integrity. It concludes with some additional benefits that may be realized with some of these methods and proposes a performance and security enhancement for the minidb system.

2. Disk Storage Systems

Disks can fail when a single bit or few bits will flip. This problem can often be detected and corrected at the hardware level by using error correcting codes in the embedded system of the drive. More extensive permanent damage to a drive can occur with a head crash or a media scratch. Mechanical failures can affect a drive motor or arm and electrical problems can damage the in-drive circuits [3].

Disk drive firmware can contain upwards of 400,000 lines of code. Firmware problems can result in permanent or transient block corruption or performance problems. Errors can also occur in the transport, the bus controller, or in the low-level software drivers. Figure 1 from Prabhakaran, Bairavasundaram, Agrawal, Gunawi, A. ArpaciDusseau, and R ArpaciDusseau [3] illustrates the complexity of the layers of the storage system. They classify disk failures as occuring at three levels: the fail stop which renders the entire disk unavailable, a block level failure, and block corruption where the individual data within a block has been altered.

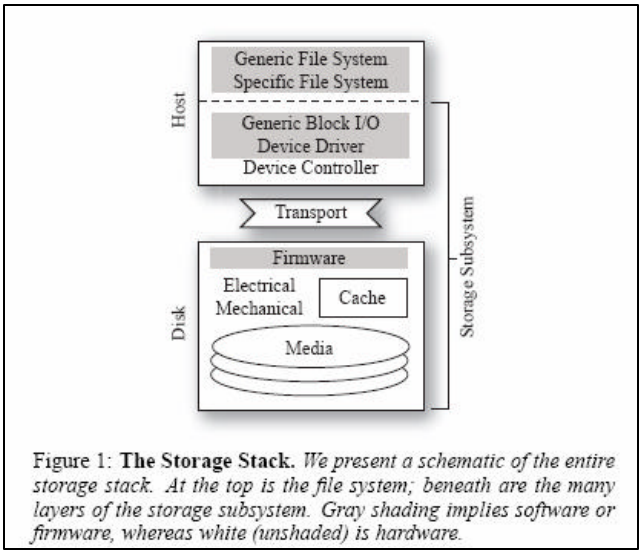


Figure 1: **The Storage Stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

Figure 1: Illustration from Prabhakaran et al. [3]

Database integrity relies on being able to design prevention, detection, and correction strategies to overcome these vulnerabilities and to accomplish that in a way that will still maximize system availability and performance.

2.1 RAID Disk Technology

The design of RAID technology [4] has allowed improvement in storage performance, reliability and recovery. The levels of RAID organization divide the disks into reliability groups with each group having extra check disks containing redundant information. When a disk fails, the assumption is that within a short time the failed disk can be replaced and the information can be reconstructed on the new disk using the redundant information. The RAID designs offer varying levels of cost and I/O performance.

RAID level 1 mirrors each disk with a full image copy. Every write to a disk is also a write to a check disk. Although the check disk can be used to improve read performance, this is a costly option. Although the integrity of the database can be checked by comparing the copies, there may not be an indication of which copy is good unless more than two copies are maintained. Both user errors and malicious changes may be replicated on all copies of the data. Possibly the strongest advantage of this RAID level is the ability to immediately substitute the mirror copy in the case of a catastrophic problem.

RAID level 2 bit-interleaves the data across the disks in a group and adds enough check disks to correct a single error. The error correction at level 2 is based on the Hamming code algorithm which is discussed in section 3. For a group of 10 data disks, this requires 4 check disks which is a cost saving improvement on the storage requirement from level 1. For a large data transfer operation, performance may improve because the controller can transfer the I/O in parallel across several disks but unfortunately, for small data transfers, performance can be hindered because all of the disks in a group must be accessed for every I/O.

The third RAID level takes into account that most disk controllers can detect which disk has failed and use the parity of the remaining good disks to reconstruct data after a failure. This decreases the reliability overhead cost at the third RAID level.

The fourth level was designed to bring down the cost of small disk transfers by striping the data across the array at the sector level instead of at the bit level. This reduces the write access requirement to two disks, a data sector and the parity sector from the check disk. This strategy suffers from a bottleneck caused by the number of accesses required of the single check disk.

As shown in Figure 2 from [5], RAID level 5 improves upon level 4 by distributing the data and the check information by sectors over all of the disks in a group. This design improvement allows for all of the disks in a group to be used to distribute the data access load and removes the bottleneck of having only a single check disk.

RAID level 6 adds an additional set of parity information on each drive. This allows a RAID 6 array of disks to recover from two simultaneous disk failures for a critical application. The cost of RAID level 6 is a decrease in performance and an increase in the storage requirement compared to level 5 [6].

Using large numbers of smaller capacity disks will tie up less of a database during the reconstruction that would be required after a failure. Note that level 5 requires that all of the disks in a group participate in an offline recovery operation while the advantage of level 1 is that it only requires the single mirrored disk.

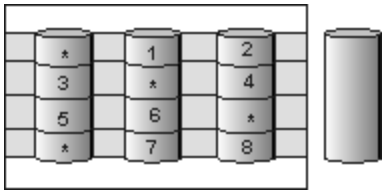
The RAID strategies allow up to two disk failures to be recovered but were designed to be used with fail stop disk technology. They also require the cost of multiple disks to provide the necessary redundancy.

2.2 Current Storage System Research

Storage systems are moving toward the use of lower cost disks that do not follow the fail stop model which causes the disk to stop operating if there is a hardware error. An undetected fault can lead to further data loss so that danger needs to be balanced against the considerable costs that can be incurred in both performance and storage utilization to provide an acceptable level of error detection and correction.

RAID technology is not an option in the PC market where the standard is a computer with one disk drive. It is estimated that to add a second disk to these systems would cause the price to increase one hundred dollars. Ongoing research is investigating ways to improve the current

In RAID level 5, the parity blocks (*) contain a representation of the data from the blocks in the same stripe.



If a drive fails in the array, the data from the failed physical drive can be reconstructed onto the hot-spare drive.

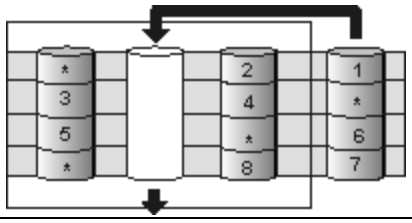


Figure 2: RAID level 5 design improvement [5]

system's ability to detect and recover from errors within the constraints imposed by a single disk drive.

Vijayasankar, Sivathanu, Sundararaman, and Zadok [7] propose an improvement to the current systems which incorporates the use of error correcting codes and remapping of bad blocks and which distinguishes blocks by their relative importance on the disk . A higher level of importance is given to reference blocks which they define as blocks that impact the ability to reach other disk blocks. They call their system *Self-Recovering Disks* (SRDs).

The SRD stores the checksum of each disk block and computes a comparison checksum during each block read. Their design is able to distinguish reference blocks as blocks which contain outgoing pointers to other blocks. When a reference block is created, its contents are replicated on the same disk but in a physically separated location to provide a higher level of insurance against a locally contained error.

Their method uses the MD5 algorithm (explained in section 3) to provide a collision resistant checksum for all blocks. The checksum is updated as each block is written to disk and recomputed and verified on a read operation. In particular if there is a problem in a reference block, there are three redundant pieces of information available; the original data block, the replicated data block and the checksum. If two of the three agree, the data can be reasonably recovered. They propose that their system can be used to help protect the lower cost SATA drives which are now often used in desktop computers. They have calculated only a 1 – 5% performance overhead as compared to traditional disks.

They also have compared their method which works at the disk level as a lower cost alternative to a software solution such as the IRON file system described below which also computes checksums for all blocks, replicates meta-data blocks and provides recovery for corrupted or inaccessible blocks [3].

Another higher level approach to this problem is to put more responsibility at the software level of the file system. File systems traditionally could trust disks to work correctly or to fail completely. Specifically, it is again the newer and less expensive SATA disks which do not guarantee fail stop operation and which are increasing in use for not only desktop computers but for large-scale computers and data storage systems.

Prabhakaran et al. [3] have classified failure detection and recovery techniques in three open source file systems, ext3, ReiserFS, and IBM's JFS and as much as possible in Windows NTFS, a closed source system. They designed their test workload to include the Posix API calls, functions such as path traversal, and files of sufficiently large sizes to exercise special features such as triple-indirect pointers. They stressed the systems by injecting faults such as block failures or data corruption and targeted both data blocks and reference block such as inodes.

In general, they made specific observations of inconsistency in failure policy, errors in implementing the policy, and inability to deal adequately with partial disk failures. Based on their analysis, they have further proposed an improved IRON (Internal RObustNess) file system. Their system addresses the same issues as the self-recovering disks but at the level of a software solution which places responsibility on the file system software to keep a checksum for all metadata and data blocks, to replicate all of the metadata blocks, and to use parity-based redundancy to protect data.

3. Database Tools

3.1 MySQL Database

Database management systems provide many administration tools to ensure the security and integrity of a database. The following discussion refers to examples from the MySQL database since it is a widely-used open source database [8].

To support the security of the database system, MySql incorporates a privilege system to ensure that users may perform only the operations allowed to them. A user's identity is determined both by their username and host from which they connect. Privilege levels can be specified at the operation level such as SELECT, INSERT, UPDATE, and DELETE. As an additional security measure user passwords are hashed and stored in a 41-byte field (a '*' and a 160-bit value based on a SHA-1 hash).

Damaged database tables can result from many sources including the hardware and software failures discussed above, improper shutdown, or file manipulation errors. A

database administrator has many commands available to safeguard the database tables including:

- CHECK TABLE can be used at several levels to verify that a table was closed properly through doing a full key lookup for all keys to ensure 100% consistency.
- CHECKSUM TABLE to report a table checksum. The CREATE TABLE statement option CHECKSUM will maintain a live checksum for all rows. There is a small performance cost but it allows corrupted tables to be identified quickly.
- REPAIR TABLE to repair a possibly corrupted table.

In addition, there are many system administration tools provided with MySQL that combine these commands into useful utility programs such as:

- mysqlcheck that checks, repairs, analyzes, and optimizes tables.
- mysqlbinlog that can read the binary log statements to help recover from a crash.
- mysqldump that dumps a MySQL database into a file as SQL, text, or XML
- mysqlhotcopy that can be used while the server is running to makes backups of MyISAM tables.

Traditionally, MySQL had been designed to put the burden of incoming data validation on the application code. This philosophy has changed since version 5.0 to give an administrator an additional tool to specify an "sql_mode" variable which can choose server-enforced data integrity. The database will then validate incoming data and reject data that is the wrong datatype or data that violates basic integrity rules (such as a date of November 31st).

In addition, if InnoDB tables are specified, foreign key constraints can be specified and enforced by MySQL. Table inserts or updates will be rejected if a foreign key value is specified that does not have a matching key value in the referenced table. If a referenced key value is to be deleted or updated and there are matching foreign key values, MySQL provides the user with action options to choose such as CASCADE, SET NULL, or NO ACTION.

System design choices are available to ensure the required level of protection for an application. An application can use the protection of ACID-compliant transactions if they choose an InnoDB storage engine. This protection costs CPU cycles and disk space so MyISAM tables can be the choice if transaction protection is not required and performance is the deciding factor.

3.2 Tools for Backup and Recovery

In a production environment, there are many strategies at the database level that are recommended as best practices to maximize the availability of a database.

Advice from the IBM website strongly encourages database administrators to create offline backups on a regular schedule to provide periodic checkpoints and to combine that with the journaling options that are available to ensure that data is not lost [9]. This strategy can be used in the case of a failure while an update is taking place by using the journal to reapply or roll-back a transaction. It can also recover if a database is corrupted, if for example a hardware failure occurs, by reinstalling the backup copy and applying the journal from that point forward.

As another example, Microsoft SQL Server™ 2005 software includes the ability to maintain a mirror database that is kept up-to-date with the production database, provides ways to perform automatic as well as manual failover in an emergency, and allows the mirror database to be located at a remote data center [10] to ensure a means for disaster recovery.

In choosing the database storage configuration, an administrator must consider not only the long term safety and recovery of the data but also the amount of system down time that may be required to restore a database to a consistent state.

Lennie et al. [11] patented an algorithm to check a large and/or replicated database by forming a checksum for each entry in the database and then using an exclusive OR to combine the individual record checksums to form a database checksum. They propose that this checksum could be recomputed periodically to ensure that the entries of the database have not been corrupted. They also propose that this checksum would be maintained at each legitimate database update by exclusive ORing the checksum of the entry both before and after modification with the database checksum. In a distributed environment, the master database checksum could be used to ensure periodically that all nodes are synchronized with the master database. Specifically, they also recommend that this check could quickly ensure that a replicated database was available to provide a backup in the case of a system failure.

4. Error Correcting Codes – Hamming

RAID level 2 storage depends on the binary code developed by Richard Hamming in the 1940s and 1950s [12]. The code is able to correct any single error in a sequence of bits and to detect a double error. Check bits are interleaved with the data bits at the positions with numbers that are a power of 2. The check bit at position 2^k checks bits in all positions which have bit k set equal to 1 in their binary representation. The value of the check bit is determined to make the parity of those bits even.

The tables from Wagner's book, *The Laws of Cryptography,* [12] are reproduced in Figure 3. The table which he labels as Table 6.1 shows the parity bits 1, 2, 4, 8, and 16 and indicates which data bits that they would check. Table 6.2 shows the check bit values in positions 1, 2, 4, and 8 for the data value 1101101. Table 6.3 illustrates that

a bit error in position 11 would cause the 1, 2, and 8 positions to be incorrect. This points exactly to the incorrect value at the position of 1 + 2 + 8 = 11.

5. Checksums and Secure Hash Algorithms

5.1 Overview

Checking the integrity of the information in the storage system is essential. The use of checksums is a well accepted way of ensuring data integrity [13]. Checksums may be able to detect data corruption due to a hardware malfunction that could otherwise go unnoticed and cause further damage. They are routinely used to validate data that must travel over network links. They can guard against malicious modification of data. They can be used to check for changes in metadata access time or modify time fields which could detect a breach in confidentiality even if file data has not been modified. Several cryptographic hash functions that are designed to be collision resistant are required by government standard.

Computing, storing, or retrieving a checksum needs to be done in a critical section of a file read or write to ensure integrity [1]. The calculation and storage decisions for checksums can affect system performance. Checksumming can be performed at the byte, block, page, or file level [1]. If the granularity of the data that is checksummed is too fine, too many computations will be required. At the other extreme, large granularity will cause additional I/O for applications which rely on small reads because the entire amount of data for the integrity check will need to be read. In general, network traffic is checksummed at each request. RAID systems perform their physical redundancy checking at the block level. Integrity checking implemented at the file system level often operates on pages. Checksumming by programs such as Tripwire [14], an open source application level security and data integrity tool that is used for monitoring and alerting if specific file changes occur, operates on a file level.

5.2 Cyclic Redundancy Check

A cyclic redundancy check is a type of hash function that leaves the data intact and appends a checksum to it. It is often used in network traffic because the recipient of the message can easily recompute the checksum to confirm the correctness of the data received.

The number calculated is a 16-bit unsigned number so there is a 1 in 65535 chance of an error not being detected because two files would have the same checksum. The CRC-16 is able to detect all single errors, all double errors, all odd numbers of errors and all errors with burst less than 16 bits in length. In addition over 99% of other error patterns will be detected [15].

5.3 Secure Hash Algorithms

A hash function H is a transformation that takes an input x and returns a fixed-size string called the hash value. In cryptography and in the government standards for secure hash functions, the requirements are more strictly defined as:

- The input can be of any length.
- The output has a fixed length.
- H(x) is relatively easy to compute for any given *x*.
- H(x) is one-way.
- H(x) is collision-free.

A hash function is "one-way" if you begin with a hash value h, it is "computationally infeasible to find some input x such that H(x) = h". A hash function is "strongly collision-free" if it is "computationally infeasible to find any two messages x and y such that H(x) = H(y)" [16].

These requirements that are defined for a secure hash function mean that if you download, copy, or receive a file; you can use the secure hash value to guarantee that you have the correct, unaltered data by comparing its hash with the original.

The MD5 algorithm was developed by Professor Ronald L. Rivest of MIT in 1994 as a way to verify data integrity that would be much more reliable than a checksum. It takes a message of an arbitrary length and produces a 128-bit "message digest" also called a fingerprint. It was developed for digital signature applications.

The MD5 algorithm was classified as a secure hash function which means that it is "computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest" [17]. It was later shown that this function was not collision-free and it was improved in the series of currently used algorithms called Secure Hash Algorithms.

These algorithms were developed by the National Institute of Standards and Technology (NIST) and are specified in federal standards to be used when a secure hash algorithm is required in federal applications. The fingerprint that is produced is also non-reversible which means that although the fingerprint uniquely identifies the data, the data cannot be reconstructed from the fingerprint.

The SHA-1 hash produces a 160-bit output fingerprint for any message that is less than 2^{64} bits in length. It is acknowledged to be slower that MD5 but is considered

Position	*1*	*2*	3	*4*	5	6	7	*8*	9	10	11	12	13	14	15	*16*	17
Bin Rep	*1*	*10*	11	*100*	101	110	111	*1000*	1001	1010	1011	1100	1101	1110	1111	*10000*	10001
Check:1	*x*		x		x		x		x		x		x		x		x
Check:2		*x*	x			x	x			x	x			x	x		
Check:4				*x*	x	x	x					x	x	x	x		
Check:8								*x*	x	x	x	x	x	x	x		
Check:16																*x*	x

Table 6.1 Position of Parity Check Bits.

Position	*1*	*2*	3	*4*	5	6	7	*8*	9	10	11
Binary	*1*	*10*	11	*100*	101	110	111	*1000*	1001	1010	1011
Word	*1*	*1*	1	*0*	1	0	1	*0*	1	0	1
Check:1	*1*		1		1		1		1		1
Check:2		*1*	1			0	1			0	1
Check:4				*0*	1	0	1				
Check:8								*0*	1	0	1

Table 6.2 Determining the Check Bit Values.

Position	*1*	*2*	3	*4*	5	6	7	*8*	9	10	11	Result
Binary	*1*	*10*	11	*100*	101	110	111	*1000*	1001	1010	1011	
Word	*1*	*1*	1	*0*	1	0	1	*0*	1	0	0 (err)	
Check:1	*1*		1		1		1		1		0	1 fail
Check:2		*1*	1			0	1			0	0	2 fail
Check:4				*0*	1	0	1					- pass
Check:8								*0*	1	0	0	8 fail

Table 6.3 Correcting a 1-bit Error.

Figure 3: Illustrating Hamming Code calculation and error recovery ability [12]

stronger against an attack. A SHA-2 group of functions has also been developed which produce longer message digests ranging from 224 to 512 bits.

In cryptography, an attack is considered a successful break if is accomplished in less than a "brute force" search even if it is not a practical risk. Collisions have been found for MD4, MD5, and SHA-0 algorithms. In 2005, Wang, Lin, and Yu [18] announced that they were able to find a method to find a collision in SHA-1 that required less than 2^{69} operations whereas a brute force method would require 2^{80} operations. Even with the existence of a method, currently a collision for SHA-1 has not been demonstrated and a practical break for the SHA-2 functions has not been documented.

However, in light of the research time that is being invested in breaking the hash algorithms, on November 2, 2007, the government announced a request for candidate algorithms to be considered for a SHA-3 standard [19].

6. Additional Benefits

6.1 Security

System integrity, security, and recovery go hand-in-hand. The use of data integrity assurance techniques can enhance the security of computer systems. In the last few years the number of system intrusion attacks has increased [13]. By detecting malicious modifications in files, damage caused by an intrusion can be reduced or prevented.

Redundant information contained in mirrored files, parity checks, or secure hash functions can be used to check for file changes or for changes in metadata fields such as access time or modify time which could indicate a breach in security even if file data has not been modified.

The database checksum calculated by the methods patented by Lennie et al. [11] could be recomputed periodically to ensure that the entries of the database have not been corrupted. It is also proposed that this check could quickly ensure that a replicated database was available to provide a backup in the case of a system failure.

6.2 Performance

The redundant information that exists as part of a system's mechanisms to ensure data integrity can also be used to improve system performance.

Since duplicate data objects would share the same checksum value, potential duplicates could be identified. A 128-bit checksum comparison could eliminate the need to compare a set of much longer data blocks. SFSRO, a secure read-only file system, names blocks and inodes with the checksums of their contents to provide an efficient way to have access to the checksum of the contents of a disk block [1].

The RAID level disk arrays can provide opportunities to realize a performance improvement. The mirror disk in RAID level 1 can be used to improve read performance. For a large data transfer operation, RAID level 2 may improve performance because the controller can transfer the I/O in parallel across several disks. Unfortunately, at this level the effect can be the opposite for a small data transfer because all of the disks in a group must be accessed for every I/O. The fourth RAID level was designed to bring down the cost of small disk transfers by striping the data across the array at the sector level instead of at the bit level so that a write access requires only two disks, the data sector and the parity sector from the check disk. To further improve performance by reducing the bottleneck caused by the number of accesses required of the check disk, RAID level 5 distributes the data and the check information by sectors over all of the disks in a group. Although the disk configuration is targeted to ensure data integrity and recoverability, all of these RAID levels can offer potential I/O performance gains [4][5][6].

7. Conclusion

Business, research, and decision making applications are increasingly dependent upon the availability of data. The value of a database is dependent upon a user's ability to trust the completeness and soundness of the information contained in the data. It is the database administrator's responsibility to choose wisely from the available tools to safeguard the data integrity.

Database integrity problems can have many sources: hardware malfunctions, software bugs, malicious attacks, or user errors. There are current tools available to avoid, detect, and correct these problems. There is ongoing research to further improve the choices.

System performance and availability requirements must be considered and balanced by cost constraints. Some methods used to ensure database integrity may also have some additional welcome side-effects in system security and performance.

It is the methods developed to use cyclic redundancy checks and secure hash algorithms to ensure data integrity that I propose to explore further in the minidb application.

8. Minidb Implementation – Using Checksum Information in Database Methods

Section 6.2 on performance discusses methods in use which leverage the redundant information that is necessary for ensuring data integrity to also improve system performance. Specifically, the methods which incorporate checksums of data can use those checksums to compare for equality. The method can be used at any level of granularity that checksums are kept: the record, table, page, or database level.

The CRC-16 checksum is widely used in monitoring message traffic. It is able to detect all single errors, all double errors, all odd numbers of errors, and all errors with bursts less than 16 bits in length. In addition over 99% of other error patterns will be detected. It does not have the security performance of the MD5 or SHA hash algorithms but it also does not require their complexity. The CRC-16 method returns a two-byte field to validate the data in comparison to fields of length 32 to 512 bytes for the MD5 and SHA hashes. The two-byte field can easily be kept and validated at both the record and table level while developing and testing the checksum database methods outlined below for the minidb system.

8.1 Minidb Structure Overview

The structure of the minidb application is illustrated in Figure 4 and includes the methods implemented to take advantage of the CRC-16 checksum information.

A minidb table is composed of variable length records written sequentially to a data file. The record structure of the data file is described by the metafile class. The index file uses a hashed key value to provide direct access to the active records in the data file. The index file is used to store the CRC-16 checksums for the methods implemented. The table class is composed of the methods which combine information from the data, meta, and index files into a working table. The sequentialIO and randomIO classes provide the necessary file access methods.

The rel_algebra class provides a set of user functions for table creation; record insert, update, and delete operations; set operations such as union, intersection, and difference; and select, project, Cartesian product and join. The class calls the table class to implement the methods. The relational algebra class has been expanded to include the additional intersection method that takes advantage of the CRC-16 checksums.

The database_tools class has been added to the minidb system to provide administrative functions to check tables for equality and to checksum tables based on the CRC-16 values stored at the record and table levels.

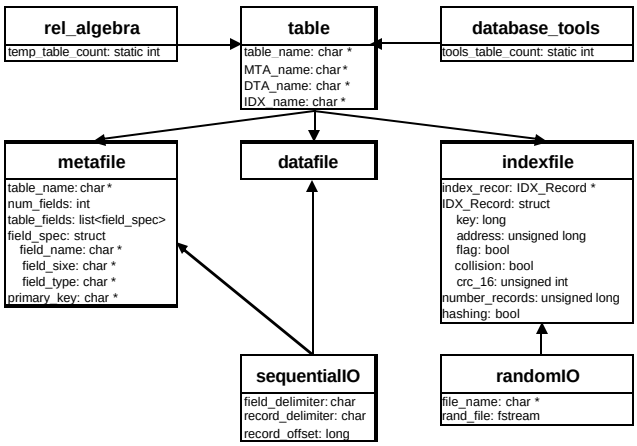


Figure 4: Minidb structure overview

8.2 The Use of Checksums for Security and Integrity

A checksum at the record level can be kept in the index file for the minidb database. The new index file record format can store the checksum of each data record as shown in Figure 5.

```
struct IDX_Record {
    long            key;        // key to search for
    unsigned long   address;    // offset in the DTA file
    char            flag;       // A = active
                                // D = deleted flag
    bool            collision;  // hash table collisions
    unsigned int    crc_16;     // checksum
    };
```

Figure 5: Modified index file record structure

A table class method calculates the CRC-16 value. The function inputs a record as a character string, processes the individual bytes, and returns a two-byte checksum for the record. Since the records are stored on disk and processed based on their field delimiters, the delimiters are included in the checksum. An online tool [20] was used in testing to check the CRC-16 results from the function.

The method developed and patented by Lennie et al. [11] to check a large and/or replicated database by forming a database checksum can be modified for the minidb application. The method proposed in [11] can be used to maintain a table checksum to determine if two tables are equivalent. This checksum test will be valid even if one of

the tables has been reorganized since it is based on the data file record contents.

To calculate the table checksum, the CRC_16 value for a table is initialized to zero when the table is created. As a record is inserted into the data file and its address and CRC-16 value are written to the index file, the CRC-16 value for the record is XORed with the CRC_16 value for the table. When a record is deleted by updating the index record active flag to a 'D', the deleted record's CRC-16 value is XORed with the table value to remove it. An update to a record is the equivalent of a delete followed by an insert.

The table checksum maintenance has been incorporated into the index file methods, and the table CRC_16 value is currently maintained in the last record in the index file.

8.3 The Use of Checksums for Query Optimization

The relational algebra methods that implement set functions can incorporate the use of the CRC-16 to check relations to find duplicates. If the two-byte CRC-16 checksums stored in the index files are not equal, the records do not have to be compared.

As an example shown in pseudo code in Figure 6, the intersection method can be modified to reduce the number of records that need to be read from disk for comparison. The greatest performance benefit would be realized in an application that did not expect to find many duplicates between large files. The crc_intersection method call has been added to the rel_algebra class and is primarily implemented in the table class.

In this method, all of the records in one index file are read to see if they are active. For each active record, the key is used to read the second index file to check for an active record. If one is found, the CRC-16 values that are stored in the index records are compared. File records are only read from disk and compared if the CRC-16 values are equal. If the file record comparison is equal, the record is written to the new intersection relation.

In comparison, the hash_intersection method reads each record of the one file and hashes the records by key into an array. It reads the second file, hashes the record key, and checks the array for an intersection. All records of both files must be read at least once for this method. If one of the files is not small enough to be hashed into memory in its entirety, it would need to be partitioned and the second file would need to be reread and compared to each partition's hash array.

In using the CRC-16 hash key value for the records, there is a one in 65535 chance that two different records would produce an equal hash key. In an application which is using a more secure hash algorithm such as MD5, SHA-1, or SHA-2; it can be reasonably assumed that if the hash value is equal, the records are the same without performing the comparison step.

```
read each record of index file 1 sequentially
for ( each active data record in file 1 )
{
    use the key to read index file 2 record
    if ( data record 2 is active )
    {
        if ( record 1 crc_16 == record 2 crc_16 )
        {
            if ( data record 1 == data record 2 )
            {
                write data record to intersection table
} } } }
```

Figure 6: Pseudo code for file intersection using CRC-16

8.4 Using Checksums for Table Comparison

In the MySQL database, if the CREATE TABLE statement option "CHECKSUM" is specified, a live checksum for all table rows will be maintained. The command CHECKSUM TABLE can be used to report the live table checksum. It can also be used to recalculate the table checksum. This tool can be used to check to see if a table has been corrupted, for example if the system was not shutdown properly. Similar tools are implemented for the minidb system in the database_tools class.

The quickest comparison implemented for the minidb is a method to compare any two tables for equality by using the table checksums as shown in Figure 7. Each time a table is reorganized, the old and new checksums can be compared as an additional way to check table integrity.

```
//    Check tables for equality using CRC-16
//-------------------------------------------------------------------
bool    check_tables_equal(char * relation2)
{
    open index file of table
    open index file of relation2 table

    get_table_crc() for table
    get_table_crc() for table relation2

    // Compare CRC_16 values returned from indexfiles
    if ( CRC-16 values are not equal )
        return (false);
    else
        return (true);
// There is a small chance of two unequal tables
//  returning the same CRC value with CRC-16.
}
```

Figure 7: Comparing tables for equality

Two more complete table checking tools are implemented in the database_tools class. The prototypes for the commands are:

- char * checksum(char * relation);
- char * checksum_index(char * relation);

These commands recalculate the value of the checksum for each active record in the file and validate it against the checksum stored in the index file. In addition, the methods recalculate and validate the table checksum. If any record checksums are found that don't match the index file checksum, the record is written to a new relation that is returned by the method.

The methods differ in the order that they read the data file. The "checksum" method reads each record of the data file sequentially. It then reads the index file record to check if the data record is active and to validate the checksum. The "checksum_index" method reads the index file sequentially and for each active record, reads the data file directly by the record address stored.

The "checksum" method may be expected to be more efficient in I/O by accessing the data file sequentially. The method can determine that a table is in error by the table checksum and can indicate the exact records in error if the errors are not part of the primary key. But it cannot specify an exact record if the error is in the key because the record key is used to access the index file; the index record for a data record with a key error will not be found.

The "checksum_index" method was written to check the table by reading the index file and checking every active record by data file address. In this way, every active record that has a CRC-16 error can be flagged. The cost of the additional information comes from accessing each data file record directly instead of sequentially as in the checksum method.

8.4 Sample Results

The performance of the CRC-16 methods was evaluated using files of up to 50,000 random records. Key values in the range from 0 to 65535 were generated using the C++ rand() function. The keys are hashed into index files which have 65536 records; a maximum of 76% of the index capacity was used.

The "crc_intersction" method was compared to the "hash_intersection" method using these files. The timing is compared by saving the value of the system clock() function before and after each method is used to intersect two tables. Sample timing code and the associated console output are shown in Figure 8 for both intersection methods.

The results from the first set of comparisons are shown in Figure 9. Sets of files containing 50,000 records each were constructed to produce intersection result sets of different sizes. The crc_intersection and hash_intersection functions were timed and the results are shown in Figure 9.

```
#include <time.h>
#include <stdio.h>

int main()
{
    clock_t before, after;
    double timing;

    rel_algebra rel_test;

    before = clock();
    rel_test.crc_intersection("new_ hash3","new_hash4");
    after = clock();
    timing = (double)after - (double)before;
    timing /= CLOCKS_PER_SEC;
    cout << "Timing for crc_hash intersection = "
         << timing <<  " sec." << endl;

    // The same code was repeated with hash_intersection
    return(0);
}
```

```
/* Console Output:

 Number of records in crc_intersection table = 3080
crc_intersection table okay: new_hash3 and new_hash4
Timing for crc_hash intersection = 1.343 sec.

Number of records in hash_intersection table = 3080
hash_intersection table okay: new_hash3 and new_hash4
Timing for crc_hash intersection = 2.625 sec.
*/
```

Figure 8: Comparing intersection methods

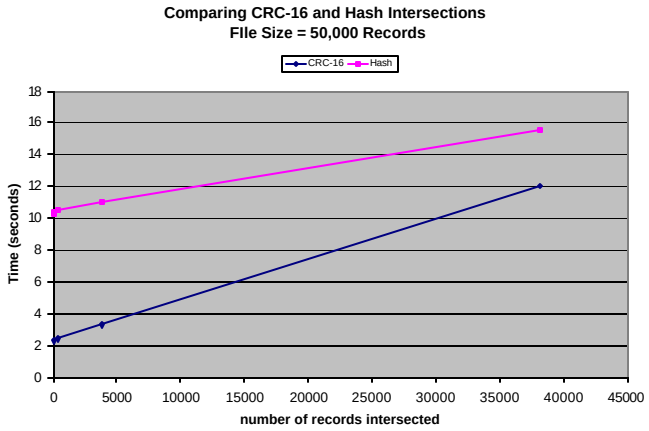


Figure 9: Time required as number of records in intersection result set increases

The CRC-16 intersection method required less time in each case tested but was approaching the hash intersection time as the number of records in the intersection set increased.

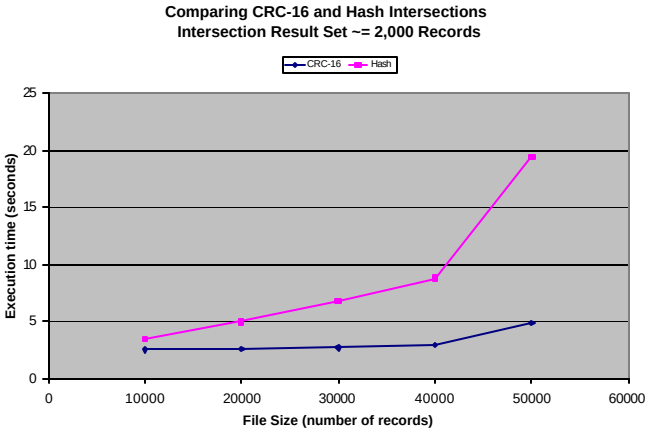


Figure 10: Comparing time required as file size increases for intersection result set of a fixed size

The second comparison, shown in Figure 10, was completed by generating files of sizes from 10,000 to 50,000 records which would contain the same intersection result set of approximately 2000 records. As expected, the time required for either intersection method increased with file size but the increase was much slower for the CRC-16 method. For a file size of 10,000 records, the CRC-16 method required 73% of the time of the hash intersection. At 50,000 records this percentage had decreased significantly, down to 25% of the hash intersection time.

As the file size increased, the CRC-16 method needed to read an increased number of index file records but the number of data file records read was held constant. In contrast, the hash intersection method needed to increase from reading 20,000 data records to 100,000 data records.

The file size of 50,000 records could still be hashed into the memory array at once. If that was not the case, the first file would have needed to be partitioned and the comparison file would have to be reread for each partition's hash array.

Table checksum test results are shown in Figure 11. Tests were run in which an error was introduced in the body of a record and in the key of a record.

The "checksum" method can find an error in the body of a record and print out the record affected. If the error is in the key of the record, the table is flagged but the record is not identified.

The "checksum_index" method can be used to identify the record in the case of a key error. The additional information comes with a time penalty. Reading the index file to identify active records and then reading the data file directly by record address took 20 – 25% longer for these test files of 10,000 records.

```
checksum("new_hash2") results:
----------------------------------------
29257^29258^29257 crc_16 not equal to index value.
Checksum error in table new_hash2
The following records were found to have a checksum error.
 newhash1    newhash2    newhash3
      29257       29258       29257
Number of records printed = 1
Timing for checksum new_hash2 = 1.203 sec.

checksum ("new_hash2") results:
------------------------------------------------
Checksum error in table new_hash2
The following records were found to have a checksum error.
 newhash1    newhash2    newhash3

Number of records printed = 0
Timing for checksum new_hash2 = 1.187 sec.

checksum_index("new_hash2") results:
------------------------------------------------
29258^29257^29257 crc_16 not equal to index value.
Checksum error in table new_hash2
The following records were found to have a checksum error.
 newhash1    newhash2    newhash3
      29258       29257       29257
Number of records printed = 1
Timing for checksum_index new_hash2 = 1.469 sec.
```

Figure 11: Comparing checksum and checksum_index methods

8.5 Conclusion

The CRC-16 functions were used successfully to improve performance in the set intersection and to demonstrate some useful administrative tools. The CRC-16 checksum provided a manageable method to test the program logic and provide some time comparisons. A secure hash function such as MD5, SHA-1 or SHA-2 could be used to improve the model.

Another improvement in the minidb implementation would be a collection of statistics for each table. At a minimum, the table checksum as well as the number of index records and data file records could be tracked. This would allow the system to increase the index file size when needed and to develop strategies to optimize queries based on choosing table methods for different file sizes.

Bibliography

[1] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok, "Ensuring Data Integrity in Storage:Techniques and Applications", *StorageSS'05,* November 11, 2005, Fairfax, Virginia, USA.

[2] James M. Slack and Elizabeth A. Unger, "A Model of Integrity for Object-Oriented Database Systems", ACM, 1992.

[3] Prabhakaran, Vijayan, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. ArpaciDusseau, and Remzi H. ArpaciDusseau, "IRON File Systems", *SOSP'05,* October 23–26, 2005, Brighton, United Kingdom.

[4] Patterson, David,, Garth Gibson, and Randy H Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, United States, 1988, Pages: 109 – 116.

[5] "Understanding RAID level-5", IBM Systems Software Information Center, http://publib.boulder.ibm.com/infocenter/eserver/v1r2/index.jsp?topic=/diricinfo/fqy0_craid5.html, last accessed November 2007.

[6] "Understanding RAID level-6", IBM Systems Software Information Center, http://publib.boulder.ibm.com/infocenter/eserver/v1r2/index.jsp?topic=/diricinfo/fqy0,_craid6.html, last accessed November 2007.

[7] Vijayasankar, Kiron, Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok "Exploiting Type-Awareness in a Self-Recovering Disk", *StorageSS'07,* October 29, 2007, Alexandria, Virginia, USA.

[8] MySQL 5.0 Reference Manual, http://dev.mysql.com/doc/refman/5.0/en/, last accessed November 2007.

[9] "Database Backup", IBM Sy stems Software Information Center,http://publib.boulder.ibm.com/infocenter/cmod/v8r3m0/index.jsp?topic=/com.ibm.ondemand.iseries.doc/ars4p05393.htm, last accessed November 2007.

[10] Mishra S., "Microsoft SQL Server 2005 Database Mirroring Best Practices and Performance Considerations", SQL Server Technical Article, February 2006.

[11] Lennie, R., C. Johnson, L. Emlich, J. Lonczak, "Method of Comparing Replicated Databases Using Checksum Information", US Patent 5974574, issued Oct. 26, 1999, http://www.patentstorm.us/patents/5784574-fulltext.html, last accessed November 2007.

[12] Wagner, Neal R., *The Laws of Cryptography,* 2003, www.cs.utsa.edu/~wagner/lawsbookcolor/laws.pdf, last accessed November 2007.

[13] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok, "Enhancing File System Integrity through Checksums", Stony Brook University, Technical Report FSL-04-04.

[14] Tripwire, Inc., "Open Source Tripwire", 2000, http://sourceforge.net/projects/tripwire, last accessed November 2007.

[15] Fairhurst, G. "Cyclic Redundancy Check", http://erg.abdn.ac.uk/users/gorry/course/dl-pages/crc.html

[16] "What is a hash Function", RSA Laboratories, http://www.rsa.com/rsalabs/node.asp?id=2176

[17] "MD5: Introduction", http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html,last accessed November 2007.

[18] Wang, Xiaoyun, Yiqun Lisa Yin, and Hongbo Yu, "Finding Collisions in the Full SHA-1", Advances in Cryptology -- Crypto'05, http://www.infosec.sdu.edu.cn/paper/sha1-crypto-auth-new-2-yao.pdf, last accessed November 2007.

[19] DEPARTMENT OF COMMERCE National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family", Federal Register / Vol. 72, No. 212 / Friday, November 2, 2007 http://www.csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf, last accessed November 2007.

[20] Bies, L., "On-line CRC calculation and free library", http://www.lammertbies.nl/comm/info/crc-calculation.html, last accessed December 2007.