SQCK: A Declarative File System Checker

Haryadi S. Gunawi, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

The lowly state of the art for file system checking and repair does not match what is needed to keep important data available for users. Current file system checkers, such as e2fsck, are complex pieces of imperfect code written in low-level languages. We introduce SQCK, a file system checker based on a declarative query language; declarative queries are a natural match for the cross-checking that must be performed across the many structures of a file system image. We show that SQCK is able to perform the same functionality as e2fsck with surprisingly elegant and compact queries. We also show that SQCK can easily perform more useful repairs than e2fsck by combining information available across the file system. Finally, our prototype implementation of SQCK achieves this improved functionality with comparable performance to e2fsck.

1 Introduction

Access to data is critical for both business and personal users of computer systems. Data is often either priceless or very expensive to re-obtain if lost; downtime and data loss combine to cost companies and end-users billions of dollars each year [22, 29]. As the central repository for much of the world's data, file systems play a central role in data protection and management. Thus, file systems should be robust and reliable.

A key component to a robust file system is a robust offline file system checker. Tools such as fsck have existed for many years [24] and are applied to restore a damaged or otherwise inconsistent file system image to a working and usable state. Although many newer file systems have tried to avoid the inclusion of an offline checker in their tool suite [19] (for example, by assuming that journaling always keeps the file system consistent), they inevitably find that a checker must be deployed. For example, SGI's XFS was introduced as a file system with "no need for fsck, ever," but soon found it necessary to deliver such a tool [15].

Unfortunately, robust checkers are not currently straightforward to design or implement. First, checkers are large and complex beasts; for example, the Linux ext2 checker contains more than thirty thousand lines of C code, while the ext2 file system itself is less than ten thousand lines. Checkers are often written in a low-level systems language such as C, which can be difficult to reason about. Checkers also are hard to test, given the huge possible state space of input file systems. Finally, checkers are often run only when a serious problem has occurred; it is well known that rarely-run recovery code tends to be less reliable [9, 28].

Given these realities, it is perhaps not surprising that file system checkers often corrupt or lose data [31, 32]. Recent work in model checking has found many serious implementation flaws in checkers, including invalid write ordering, buggy transaction abort, incorrect optimization, and unattempted recovery of invalid directory entries [31, 32]. Our evaluation of e2fsck, the Linux ext2/3 checker, confirms that it has many weaknesses. In particular, e2fsck sometimes performs inconsistent repairs that can corrupt the file system image by overwriting important metadata (including the superblock); e2fsck also sometimes does not use all available information and can lose portions of the directory tree.

To build a new generation of robust and reliable file system checkers, we believe a new approach is required. The ideal approach should enable the high-level intent of the checker to be specified in a clear and compact manner; further, the description of the intent should be cleanly separated from its low-level implementation and how it is optimized. A high-level specification has multiple benefits: by its very nature it is easier to understand, modify, and maintain.

In this paper, we introduce SQCK (pronounced "squeak"), a novel file system checker. Borrowing heavily from the database community, SQCK employs declarative queries to check and repair a file system image. We find that a declarative query language is an excellent match for the cross-checks that must be made across the different structures of a file system.

Our experience shows that declarative repairs can be surprisingly elegant and compact, especially compared to the original e2fsck code. Specifically, we find that SQCK can reproduce the functionality of e2fsck in many fewer lines of code; the SQCK checks and repairs require only about 1100 lines of SQL (along with some helper code written in C).

We find that SQCK can improve upon the traditional checks and repairs as well. First, SQCK avoids the inconsistent repairs performed by e2fsck by ensuring that its queries are executed in the correct order; specifically, a file system structure is only repaired after the location of that structure has been validated. Second, SQCK can perform more interesting and complete repairs than e2fsck by combining information from multiple sources. For example, SQCK performs majority voting over superblock and group descriptor replicas to handle the case where the primary copy is corrupted. SQCK also examines the ".." entry of a directory to verify the correct parent when there is conflicting information. Finally, SQCK ensures that its repairs follow the same allocation policies as ext2 by laying out new blocks with the appropriate locality.

SQCK achieves this simplicity and completeness with no cost to performance. Our evaluation of the first-generation prototype of SQCK on top of the MySQL DBMS [1] shows that SQCK can handle even large file system partitions with comparable performance to e2fsck. Overall, we believe that the SQCK-style declarative approach will lead to a new generation of simpler, more robust, and more complete file system checking and repair.

The rest of this paper is organized as follows. In Section 2, we present background information on the state of the art of checking and evaluate a traditional file system checker. We present the design and implementation of SQCK in Sections 3 and 4 and then evaluate SQCK in Section 5. We then discuss related work in Section 6 and conclude in Section 7.

2 Fsck Background

To create a better file system checker, one first needs to understand the current state of the art. In this section, we give a brief overview of the checks and repairs performed by e2fsck for an ext2 file system [10]. We then describe in detail the weaknesses and non-optimal repairs performed by e2fsck. Finally, we explain why modeling languages [11, 12, 21] are not as suitable as declarative query languages for file system checking.

2.1 Fsck Overview

Despite the best efforts of the file and storage system community, file system images become corrupt and require repair. While it is obvious that non-journaling file systems (*e.g.,* ext2) can easily become inconsistent due to untimely crashes, other file systems can as well. In particular, problems with many different parts of the file and storage system stack can corrupt a file system image: disk media, mechanical components, drive firmware, the transport layer, bus controller, and OS drivers [6, 7, 17, 18, 27, 30]. Since file systems do not usually contain the machinery to fix corruptions themselves [8, 27], there is a broad need for robust file system checkers.

Given both its popularity and our ability to access its source code, we focus on the file system checker for ext2/ext3, e2fsck. The purpose of the e2fsck utility is to check and repair the data structures of an ext2/ext3 file system on disk; in the ideal case, the repaired file system is readable, writable, and contains all of the directories, files, and data of the original file system.

e2fsck is a non-trivial piece of code: it contains more than 30,000 lines of C code and can identify and return 269 different error codes. Its checks and repairs are performed in six different phases [24], in which scanning the disk, checking the data structures, and repairing any inconsistencies are all intermixed. Many of the simplest checks examine individual structures in isolation (*e.g.,* that superblock fields, inode fields, and directory entries all appear valid) or verify that pointers fall within the expected ranges. More interesting and costly checks validate that no two pointers (*e.g.,* across all inodes) point to the same data block. Other intensive checks peruse the file system tree, ensuring that all files and directories are properly connected.

2.2 Fsck Weaknesses

To understand the weaknesses of e2fsck, we need to understand the individual repairs performed by e2fsck in response to different errors it encounters. We are not explicitly interested in finding implementation bugs [31, 32], but in understanding when e2fsck could have made better repairs than it did for a given corruption.

2.2.1 Fault Injection Methodology

To begin to understand the complex runtime behavior of e2fsck, we explore how e2fsck repairs a single on-disk corruption. Given that it is infeasible to exhaustively corrupt every data structure field to every possible value, we limit our scope to corrupting on-disk pointers. Ext2 contains two classes of pointer. First, a block pointer contains a physical block number; for example, data block pointers in inodes contain the block numbers of corresponding data blocks. Second, an index pointer contains an index into a table; for example, an inode index picks an entry in the inode table within a block group.

We use our knowledge of ext2 to further reduce the search space. In particular, we corrupt pointers in a type- and location-aware fashion [8]. Specifically, we assume that the e2fsck repair depends only on: (i) the type of pointer that has been corrupted, and (ii) the type of block that it points to after corruption and whether it lives in the same or a different block group. For example, (i) corrupting File A's data pointer is the same as corrupting

File B's data pointer, and (ii) corrupting a pointer to refer to inode-block P in group G is the same as corrupting it to refer to inode-block Q in group G.

To corrupt the file system and examine the results, we use the debugfs utility [2]. We corrupted approximately 10 different pointers to 18 different values for a total of 180 corruption tests. To the best of our knowledge, all of our findings are new.

2.3 Results

From our fault injection experiments, we find that e2fsck fails along four different axes. First, e2fsck does not always create a *consistent* file system, even though this is the explicit purpose and goal of fsck. In fact, in some cases, e2fsck will perform an imprudent "repair" that transforms a file system with a relatively small inconsistency into one that is completely unreadable.

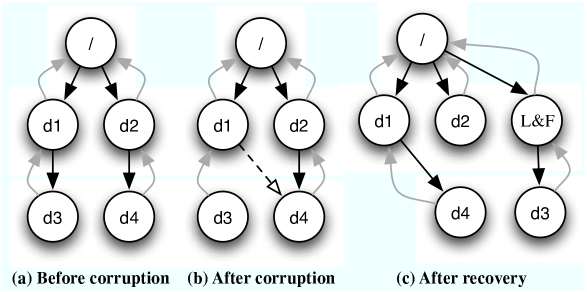
Second, e2fsck does not always perform an *information-complete* repair. We define a repair to be information-complete if it reconstructs the file system to match the original file system to the greatest extent possible given the information available on disk. The notion of an information-complete repair is needed because a repair can easily create a consistent, but useless file system by simply removing all of the contents. For example, an information-complete repair should always incorporate redundant copies.

Third, e2fsck does not always perform a *policy-consistent* repair. We define a repair to be policy consistent if it follows the same policies as the original file system; for example, since ext2 allocates data blocks in the same group as its corresponding inode, e2fsck should as well.

Finally, e2fsck does not always perform a *secure* repair. Specifically, e2fsck sometimes leaks information from one data structure to another when it clones blocks. In this way, it is possible for a user's file to be "repaired" to contain data from a file in the root directory.

We now describe the specific behavior of e2fsck that leads to these problems.

Inconsistent Repair: *Clears "Indirect Blocks" Incorrectly.* Fundamentally, e2fsck checks and repairs certain pointers in an incorrect order; as a result, e2fsck can itself corrupt arbitrary data on disk, even the superblock. Specifically, e2fsck clears block pointers that fall out of range of the file system inside indirect blocks without first checking that the pointer to the indirect block itself is correct. Thus, if an indirect pointer was corrupt, e2fsck may clear the block that the indirect pointer incorrectly refers to. This clearing can lead to arbitrary corruptions of file, directory, and meta-data in the file system; most notably, if the file system contains only a single superblock, the file system can even be unmountable after running e2fsck.



(a) Before corruption **(b) After corruption** **(c) After recovery**

Figure 1: **The false parenthood problem.** *This figure shows the problem in the recovery done by e2fsck for corruption in directories. Each node is a directory in the file system. For clear understanding, we use dotted backpointers to show the parent for each directory as present in the '..' entry for that directory. Part (a) of the figure shows the initial file system structure. Part (b) shows the file system structure after corruption. We inject this fault where the entry for dir3 in dir1 is corrupted to point to the inode of dir4. After recovery by e2fsck, the dir1 claims dir4 and the original parent child link between dir2 and dir4 is deleted. This results in totally different structure of the file system after recovery as shown in part (c). For convenience we show the lost+found (L&F) directory only in the final structure.*

Information Incomplete: *False Parenthood.* e2fsck does not always use all of the information available to it regarding directories. One example is the case where an inode index within a directory is corrupted to point to a different valid directory inode. This situation is illustrated in Figure 1. If a directory entry is corrupted to point to another target directory (parts a and b), the e2fsck repair might move the target directory to the wrong parent (part c).

We emphasize that enough information is available in an ext2 file system for e2fsck to make the correct repair: each directory contains an entry for its parent (denoted ".."). To perform an information complete repair, e2fsck could simply observe this entry to keep the target directory with its correct parent and to reattach the lost directory to its parent instead of moving it to lost+found. In general, the directory hierarchy in ext2 contains much more information than is being used currently in e2fsck.

Information Incomplete: *Ignores Replicas of Inode Table Pointers.* Ext2 contains replicas for important meta-data, such as pointers to the inode tables; however, e2fsck does not always use this redundant information. For example, when an inode table pointer becomes corrupted and points to other blocks inside the same block group, e2fsck assumes the pointer is correct; e2fsck then finds that the "inodes" are not valid. For consistency, e2fsck removes the corresponding directories and files from the directory tree; if this group contains the root directory, the file system is trivially consistent with no directories. Again, enough information is available for e2fsck to make the correct repair: each inode table pointer is replicated across block groups; e2fsck should check that all block groups agree on these important values.

Policy Inconsistent: *Different Layout.* e2fsck does not allocate blocks on disk with the same layout policy as ext2; as a result, e2fsck can fragment files and directories, degrading the future performance of file system operations. For example, when e2fsck detects that the same data block is pointed to by both a directory and a file, e2fsck clones the block by allocating a new block for the file and retaining the old block for the directory. To perform a policy-consistent repair, e2fsck should allow the closer inode to retain the original data block.

Insecure Repair: *Copies Data Freely.* Whenever e2fsck discovers that two pointers refer to the same block, e2fsck clones the block. However, this policy has the potential to leak private information. For example, if a data block is shared by two inodes, one in the `/home/userA` directory and one in the `/root` directory, we might want to remove the pointer from `userA` and keep the one from the root.

Summary: We have found that e2fsck has a number of problems in how it performs repairs; we note that these problems are not simple implementation bugs, but are fundamental design flaws. In particular, it is difficult for e2fsck to combine the many pieces of information available (*e.g.*, replicas of pointers and parent directory entries) and to ensure that all checks and repairs are done in the correct order.

2.4 Other Approaches

Given the difficulties of implementing a file system checker, an alternative is needed. File system checking can be viewed as ensuring that the content satisfies a specification; therefore, some researchers have attempted to auto-generate fsck code by writing a specification in an object modeling language. Specifically, Demsky and Rinard’s work repairs inconsistencies automatically given specified constraints [12]. Their automated repair finds the cheapest way to repair the system such that it satisfies the constraints again. For example, if two inodes share the same data block, the cheapest repair could simply remove one of the pointers; however, this may not be the desired result. In fact, there are many ways to solve the problem: the inode with the earliest modification time could release the block [24], the block could be cloned (e2fsck’s way), or the operator could decide. In our terminology introduced above, previous approaches ensure that the repairs are consistent, but not necessarily information-complete or policy-consistent.

When reinventing fsck, we need a language that can declaratively express both the checks and the repairs. Like others who have applied declarative languages to domains such as system configuration [13] and network overlays [23], we believe that the solution is to use a declarative query language. Declarative query languages have been built from day one to both cross-check and update massive amount of data. Hence, we believe utilizing a declarative language is a better fit than a specific modeling language for fsck.

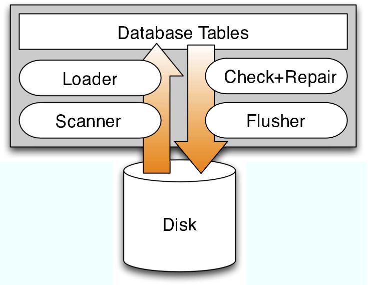


Figure 2: **Architecture.** *The diagram depicts the basic SQCK architecture. The left part of the design, the loader and scanner, and the right part of the design, the checker and flusher are decoupled, allowing us to optimize each component in isolation.*

3 Designing SQCK

In this section, we describe the design of SQCK, our file system checker based on declarative queries. We first describe our goals and then present the overall architecture of SQCK, including how declarative checks and declarative repairs are performed. We then describe three simple versions of SQCK: one that emulates e2fsck, one that fixes the inconsistent repairs of e2fsck, and one that significantly improves the types of repairs performed.

3.1 Goals

We believe that a file system checker should be correct, flexible, and have reasonable performance; we believe a declarative language will enable us to meet these goals for the following reasons.

Correctness: The primary responsibility of a file system checker is to produce a consistent file system image. A declarative language allows one to check and repair hundreds of corruption scenarios in a clean and compact fashion; we believe the ability to produce correct repairs is improved due to the simplicity of the queries and the separation of the specification from the implementation. A secondary goal is to produce repairs that leverage all of the on-disk information to retain as much as possible of the file system. We believe declarative languages allow one to easily combine the disparate information that resides throughout the file system.

Flexibility: Given a single corruption, there are many reasonable repairs that could be performed. The simplicity of a declarative language encourages one to explore

Tables	**Fields**
Superblock Table	*blkNum*, *copyNum*, *dirty*, firstBlk, lastBlk, blockSize, ...
GroupDesc Table	*blkNum*, *gdNum*, *copyNum*, *dirty*, *start*, *end*, blkBitmap, inoBitmap, iTable, ...
Inode Table	*ino*, *blkNum*, *used*, *dirty*, mode, linksCount, blocksCount, size, ...
DirEntry Table	*blkNum*, *entryNum*, *dirty*, ino, entryIno, recLen, nameLen, name
Extent Table	start, end, *pBlk*, *pByte*, *type*, *startLogical*, *endLogical*, *ino*, *dirty*, ...

Table 1: **SQL Tables.** *Italic fields represent information we generate since they are not stored on the disk.*

this policy space and even provide different modes of repair (*e.g.*, fast but partial repair, or slow but full/smart repairs).

Performance: While the performance of a file system checker is not a primary concern, it must not be prohibitively slow; specifically, the checker must be able to handle the amount of data on modern disks and storage systems. Thus, our goal is to create a checker that is competitive in speed to the original e2fsck.

3.2 Architecture

SQCK contains five primary components, as shown in Figure 2. The *scanner* reads the relevant portions of the file system from the disk, while the *loader* loads the corresponding information into the database *tables*. The *checker* is then responsible for running the declarative queries that both check and repair the file system structures. The *flusher* completes the loop by writing out the changes to disk. We postpone our description of the scanner, loader, and flusher until Section 4. In this section, we explain the tables and the checker.

3.2.1 Database Tables

It is important to construct the database tables such that the SQCK checker can perform efficient queries that cover the same repairs as e2fsck. Conceptually, SQCK contains a table for each of the different metadata types in the file system: superblocks, group descriptors, inodes, directories, and block pointers [10]. Together, the tables store all of the information about the file system image that was originally on disk. However, with this on-disk information alone, the SQCK checks and repairs are neither simple nor efficient; therefore, SQCK stores extra, easily calculated information in the tables. Table 1 shows the five database tables utilized by SQCK. We describe briefly the important fields in each table.

Superblock: Since the superblock is replicated, we load each replica into a row of the table; this table allows SQCK to easily check the consistency across superblocks. As expected, each row contains the information available from the superblocks on disk. To be able to reflect repairs back to the disk in the flusher, we also introduce *copyNum* and *blkNum* fields that specify where a replica lives on the disk and a *dirty* field.

GroupDescTable: Each group descriptor and its replicas are loaded into separate rows of this table; as expected, we store here the on-disk information such as the pointers to the block bitmap, inode bitmap, and inode table. SQCK also adds the *start* and *end* block of each group; this addition allows SQCK to easily check whether pointers fall within the desired range of the block group.

InodeTable: Each row of the table corresponds to a different allocated inode, with appropriate fields for the on-disk information such as mode, links, and size. The *used* field tracks which inodes are part of the final directory tree so that SQCK can calculate the final inode bitmap.

DirEntryTable: Each row of the table corresponds to a different directory entry. SQCK performs many cross-checks on this table to verify the directory tree structure.

ExtentTable: The conceptual idea of this table is to record all of the pointers to data blocks, so that SQCK can ensure that no two pointers refer to the same block. In our initial implementation, we loaded each direct pointer as its own row; however, this is intractable for a large file system because the table grows too large and the loader takes too long. Therefore, we switched our table design to represent extents of contiguous direct blocks; specifically, each extent specifies the start and end block. Additionally, each row records the location of the original pointer and the *type* of the pointer (*e.g.*, direct, single, double, or triple indirect).

3.2.2 Declarative Checks

A declarative query language is an excellent match for the checks and repairs that must be performed by a file system checker. To give some intuition as to why this is true, we categorize the different checks that must be made and show how a prototypical check from each category can be specified with SQL [3].

The original e2fsck performs a total of 121 interesting repairs. We have categorized all of these repairs into four categories, depending upon how many file system structures the repair must simultaneously peruse. As shown in Table 2, a repair can touch a single instance of a single structure type, one instance of one type with another of a different type, multiple instances of the same type, or multiple instances from multiple types.

There are 63 fsck repairs that involve fields of a single structure in isolation. A simple example of this type of repair is ensuring that the deletion time of a used inode is zero. Another example is verifying that the block bitmap for a group is located within that group. We show

```
SELECT *
FROM   GroupDescTable G
WHERE  G.blkBitmap NOT BETWEEN G.start AND G.end
```

Figure 3: **Check block bitmap not in group.** *This query finds a block bitmap pointer (blockBitmap) of a group that points outside the group.*

```
SELECT  X.*
FROM    ExtentTable X, SuperblockTable S
WHERE   S.copyNum =  1                  AND
        X.type    =  INDIRECT_POINTER   AND
       (X.start   <  S.firstBlk         OR
        X.end     >= S.lastBlk)
```

Figure 4: **Check illegal indirect block.** *An illegal indirect block is one that points to outside the file system range*

```
SELECT  *
FROM    DirEntryTable P, DirEntryTable C
WHERE   // P says C is his child
        P.entryNum >= 3      AND
        P.entryIno  = C.ino  AND
        // but C says P is not his parent
        C.entryNum  = 2      AND
        C.entryIno <> P.ino
```

Figure 5: **Bad dot dot.** *This query finds a directory entry that does not claim the actual parent.*

```
SELECT X.*
FROM   ExtentTable X
WHERE EXISTS
 (SELECT *
  FROM   SuperblockTable S
  WHERE
   // extent overlaps superblock copies
   S.blk BETWEEN X.start AND X.end)
 OR EXISTS
 (SELECT *
  FROM   GroupDescTable G, SuperblockTable S
  WHERE
   // or extent overlaps group descriptors
   (X.start BETWEEN G.blk AND G.blkEnd OR
    X.end   BETWEEN G.blk AND G.blkEnd) OR
   // or extent overlaps inode table
   (X.start BETWEEN G.iTbl AND G.iTblEnd OR
    X.end   BETWEEN G.iTbl AND G.iTblEnd) OR
   // or extent overlaps block bitmap
    G.blkBitmap BETWEEN X.start AND X.end OR
   // or extent overlaps inode bitmap
    G.inoBitmap BETWEEN X.start AND X.end)
```

Figure 6: **Check block overlaps metadata.** *This query locates inode's extents that overlap with the filesystem metadata. To reduce space, we abbreviate some fields: G.iTblEnd should be G.iTable + S.inodeBlocksPerGroup - 1; G.blkEnd should be G.blk + S.gdBlks - 1.*

	Single instance	**Multiple instances**
Intra structure	*Category #1* 63 checks	*Category #3* 11 checks
Inter structures	*Category #2* 12 checks	*Category #4* 35 checks

Table 2: **Taxonomy of fsck cross-checking.** *We distinguish four types of cross-check. We report the number of checks that fall into each category. In the first category, a cross-check can be made within an instance of a structure. In the second, a cross-check is performed on an instance of a structure and an instance of another different structure. The third category cross-checks multiple instances of a structure. Finally, the last category involves information stored in multiple instances of more than one structures. Each number in the box represents the number of checks that are done by e2fsck in each category.*

how this check can be expressed simply and efficiently using SQL in Figure 3. The query simply performs a SELECT from the group descriptor table to find any bitmaps that are not within the desired range for the group. Thus, range-checking queries are easily specified.

The second category includes checks between one instance of a structure and an instance of another different structure; fsck runs 12 checks of this type. A simple example is verifying that all pointers refer to blocks within the file system; this check involves verifying that every pointer is within the range specified in the primary superblock. Unlike the previous example, this example must examine values in different structures and subsequently different tables. Figure 4 shows how to check that no indirect block points outside the file system. Specifically, the query returns all extents (`X.start..X.end`) corresponding to indirect pointers that fall outside the file system range specified in the primary superblock (`S.firstBlk..S.lastBlk`). Hence, SQCK can easily join multiple structures to perform the necessary cross-checks.

The third category contains 11 cross-checks of multiple instances of the same structure. One example of this type of repair is checking that multiple inodes do not point to the same data block. A second example, shown in Figure 5 checks that the ".." entry of a directory points to the actual parent. This check can be done easily in SQL: the query simply joins the directory entry table with itself, selecting cases where the parent directory contains an entry for a child (where `P.entryNum >= 3`), but the child's entry for ".." (`P.entryNum = 2`) is not the parent's inode.

Finally, 35 checks fall into the fourth category in which the cross-checks involve multiple instances of more than one structure. One example is the rule that validates the link count of an inode, since it must traverse all directory entries and count how many times each en-

```
SELECT  P.entryIno, COUNT(*), MIN(P.ino)
FROM    DirEntryTable P, InodeTable I
WHERE   P.entryNum >= 3      AND
        P.entryIno  = I.ino  AND
        I.mode      = DIR
GROUP BY P.entryIno
HAVING (COUNT(P.entryIno) > 1)
```

Figure 7: **Check multiple parents.** *This query returns directories that have multiple parents. The parent that has the smallest inode number (*`MIN(P.ino)`*) will be the one that keeps the child directory.*

```
UPDATE  ExtentTable X
        INNER JOIN
        (Query in Figure 4) AS V
        ON X.ino   = V.ino    AND
           X.type  = V.type   AND
           X.start = V.start  AND
           X.end   = V.end
SET     X.start = 0, X.end = 0, X.dirty = 1
```

Figure 8: **Repair illegal indirect block number.** *This query repairs indirect block numbers that fall outside the file system range (returned by query in Figure 4), by clearing them to zero.*

```
result = run(findUnconnectedDir.sql);        [9]
while(dir = mysql_fetch_row(result)) {
 run(changeDotDot.sql, dir, lfIno);           [3]
 slot = run(findEntrySlot.sql, lfIno);        [7]
 if (!slot) {
  lfBlk = run(getLocation.sql, lfIno);        [3]
  newBlk = run(allocNewBlock.sql, lfBlk);  [25]
  if (run(needIndirect.sql, lfIno))           [5]
   { // alloc indirect (not shown) }
  run(addNewBlock.sql, newBlk, lfIno);        [3]
  run(addInodeSize.sql, lfIno);               [3]
  run(initNewDirBlk.sql, newBlk, lfIno);      [3]
  slot = run(findEntrySlot.sql, lfIno);       [7]
 }
 // now break the slot and prepare          [13]
 // newSlot based on dir.  (not shown)
 run(updateOldSlot.sql, oldSlot);             [3]
 run(insertNewSlot.sql, newSlot);             [3]
 run(incrementLinkCount.sql, lfIno);          [3]
}
```

Figure 9: **Complex repair.** *The C pseudo-code above illustrates the complex repair in reattaching unconnected directories to the lost+found directory. The bold texts are the SQL files that are executed. The bold numbers in the brackets represent the lines count of each SQL file. The italic number is the lines count of the C code.* `lfIno` *is the inode number of the lost+found directory.*

try appears. We give two examples of these queries to further convince the reader that even these types of seemingly complicated checks are surprisingly straightforward to express.

The first example checks for conflicting block pointers; in ext2, block pointers are stored in many places and none should refer to the same block. Figure 6 shows a query that ensures blocks pointed from an inode do not overlap with file system metadata blocks. The query is a little bit cumbersome because it checks whether an extent overlaps with each piece of file system metadata separately (*i.e.*, superblock copies, group descriptors, inode bitmaps, block bitmaps, and inode tables).

The second example verifies that multiple directory entries do not point to a same directory, corresponding to the false parenthood problem discussed in Section 2.3; we show how it can be expressed in SQL in Figure 7. Basically, the query selects directory entries that appear more than once in the tree structure. In more detail, the query does not select the "." or ".." entries and selects only directory inodes, as determined by their `mode` field in the inode table. Counting the number of entries satisfying this constraint is straightforward with the ORDER BY and HAVING features of the query language. Note that this query returns the smallest inode number among the parents (`MIN(P.ino)`), which is needed to mimic how e2fsck incorrectly repairs this problem. In particular, e2fsck always assumes the parent with the smallest inode number is the real parent without consulting the ".." entry of the child. We show how we can easily improve this query in Section 3.3.3.

3.2.3 Declarative Repairs

Performing checks of file system state is only part of the problem; after SQCK detects an inconsistency, it must then perform the actual repair. SQCK performs the repair by first modifying its own tables; the flush process then propagates these changes to the disk itself. We have found that repair operations on the tables can be performed in one of two ways.

In the simplest cases, a repair must simply adjust a few fields within a table. These repairs can be performed by embedding the declarative checks presented previously into a larger query that then sets fields within the selected rows. For example, an illegal indirect block pointer (one that points outside the file system range) is fixed by clearing the pointer to zero. Figure 8 shows that these pointers can be cleared with a query that sets to zero the illegal extents found by the check query in Figure 4. Note that the query also sets the dirty flag so that the flusher will later propagate these changes from the database tables to the on-disk structures.

In more complex cases, repairs may need to update more than one table. In these cases, SQCK combines a series of SQL queries with C code. SQCK currently supports a variety of repair primitives, such as finding free blocks and inodes and adding and deleting extents, directory entries, and inodes. Figure 9 shows how a valid directory with a reference count of zero (*i.e.*, a lost directory) is reconnected to the lost+found directory. Briefly, the code behaves as follows. After a query finds the set of unconnected directories, SQCK performs the following operations on each such directory. First, the ".." entry is adjusted to point to lost+found. Next, a directory entry slot is allocated within lost+found, which may require allocating new blocks and increasing the size of the lost+found. After the slot is ready, the entry is filled to correspond to the unconnected directory.

3.3 Possible Repair Policies

The simplicity of implementing checks and repairs in SQCK enables one to construct different versions with different repair policies. At this time, we have created three versions of SQCK: one that emulates e2fsck with both its good and bad polices, one that fixes what e2fsck does wrong (*i.e.*, fixes the inconsistent repairs described in Section 2), and one that adds new functionality that e2fsck does not even attempt (*i.e.*, performs information-complete and policy-consistent repairs). We briefly describe these three different versions.

3.3.1 Emulating e2fsck

Our basic version, SQCK$_{fsck}$, emulates the repairs made by e2fsck. From our analysis of e2fsck, we have determined that it performs 153 different repairs, of which 121 are significant and interesting for ext2 (the remaining 32 repairs fix the ext3 journal and other optional features). These 121 repairs are detailed in Table 3. As shown, e2fsck performs these repairs in six distinct phases, in which reading the file system image from the disk is intermixed with the actual checks and repairs. SQCK$_{fsck}$ implements these 121 repairs each as a separate query within the check and repair process.

3.3.2 Fixing e2fsck

Our second version, SQCK$_{correct}$, fixes the inconsistent repairs performed by e2fsck. As described in Section 2, if e2fsck follows a bad pointer to what it believes is an indirect block, it can corrupt the file system and leave it in an inconsistent state. The basic flaw of e2fsck is that it performs certain checks and repairs in the incorrect order: it wrongly "repairs" direct pointers before checking that the indirect block containing those pointers is valid.

In general, repairs of a complex data structure must be performed in a specific order; specifically, if a piece of information A is obtained from B, then B must be checked and repaired first.

#	Checks Performed
28	**Phase 0: Check consistency in the superblock**
23	*Field check:* Check all superblock fields (*e.g.*, fs size, inode count, groups count, mount/write time)
3	*Range check:* Ensure pointers to block bitmap, inode bitmap, inode table are in the group
2	*Special feature:* Check resize inode feature
35	**Phase 1: Scan and check inodes and block pointers**
9	*Bad block:* Check fields of bad-block inode; ensure superblocks and group descriptors in healthy blocks
18	*Inode structure:* Check fields (*e.g.*, mode, time, size) of different inodes (*e.g.*, root, reserved, boot load)
1	*Range check:* Ensure direct and indirect pointers point within the file system
7	*Conflicts:* Ensure no conflict among block pointers (*e.g.*, two inodes should not share a block)
38	**Phase 2: Scan and check all directory entries**
16	*Directory:* Check each has '.' and '..' entry, '.' points to itself, does not have missing block, fields of dir inode consistent (*e.g.*, acl, fragment size)
9	*Dir Entry:* Check each entry has correct name length, each points to an in-range inode, record length valid, filename contains legal characters
5	*Pathnames:* Each entry points to used inode, does not point to self, does not point to inode in bad block, does not point to root, dir has only one parent
8	*Special inodes:* Check device inodes and symlinks
6	**Phase 3: Ensure all directories are connected to the file system tree**
3	*lost+found*: Ensure lost+found directory is valid and ready to be populated
3	*Reattach*: Reattach orphan directory to lost+found
3	**Phase 4: Fix reference counts and reattach zero-linked file to lost+found**
11	**Phase 5: Check block and inode bitmaps against on-disk bitmaps**
121	**Total**

Table 3: **Repairs performed by fsck and SQCK$_{fsck}$.** *The table summarizes the 121 repairs performed by the traditional fsck.*

We have constructed an *information dependency graph* for the data structures in ext2 to ensure that repairs are performed in the correct order. A portion of this graph is shown in Figure 10. The figure illustrates that e2fsck does not follow the order specified by the dependency graph. SQCK$_{correct}$ reorders the relevant queries to ensure that single, double, and triple indirect blocks are all validated in the correct order before repairing the direct pointers themselves. We find that reordering repairs in SQCK is straightforward due to the structure of the queries; we do not believe such reordering is simple in e2fsck.

Currently, the dependency graph must be manually constructed by the file system developer/administrator. Since the repair queries in SQCK are neatly structured,

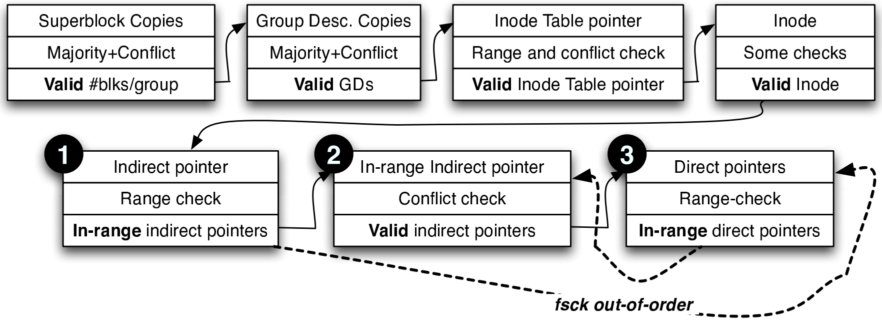


Figure 10: **Information dependency graph.** *The figure shows a chain of information dependency. Note that the full graph forms a tree-like graph; to save space, only a partial dependency chain is shown. Each box contains three rows: a new information obtained from the previous box, the check (and the corresponding repair, not shown), and the new state of the information after the check. For example, in box 1, an indirect pointer is obtained from a valid inode. After the range-check, the indirect pointer is marked in-range, but not yet valid. After it passes the conflict-check in box 2, it is finally marked as valid, which implies that we can proceed to box 3 which repairs out-of-range direct blocks contained in this indirect block. Unfortunately, e2fsck does not follow this ordering, as shown by the dashed lines; fsck proceeds repairing the direct pointers from a not yet valid indirect block. When e2fsck later finds out that the indirect block is indeed invalid (e.g., conflicting with other file system metadata), the content of the metadata has been accidentally corrupted in box 3.*

New Repair	LOC (C)	LOC (SQL)
Majority rule on block bitmap pointers	40	22
Majority rule on inode bitmap pointers	40	22
Majority rule on inode table pointers	40	22
Finding false parents	13	14
Reconstructing missing directories (*)	47	20
Precedence cloning	23	19
Secure cloning (**)	41	8

Table 4: **New repairs.** *The table lists all the new repairs we introduce. (*) In addition to the number of lines reported here, this new rule heavily uses the primitives as in Figure 9. (**) The number of lines reported for this rule is the additional code to the original cloning repair.*

```
SELECT  F.*
FROM    DirEntryTable P, DirEntryTable C,
        DirEntryTable F
WHERE   // P says C is his child
        P.entry_num >= 3      AND
        P.entry_ino  = C.ino  AND
        // and C says P is his parent
        C.entry_num  = 2      AND
        C.entry_ino  = P.ino  AND
        // but F, the false parent, says
        // C is also his child. P wins.
        F.ino <> P.ino        AND
        F.entry_num >= 3      AND
        F.entry_ino  = C.ino
```

Figure 11: **Finding false parents.** *This query returns the actual false parents. A false parent is a parent that claims to own a child even though the child is already strongly connected to another parent.*

the ordering can then be manually verified against the dependency graph. More ideally, a static tool could be built on top of SQCK to verify the ordering automatically. Specifically, each query could be tagged with a unique name that describes the check/repair performed, then a parser could automatically construct the ordering from the code, and finally a verifier could compare the constructed ordering against the specified ordering. This highlights that a structured fsck can be easily verified than a cluttered one.

3.3.3 Improving e2fsck

Our final version, SQCK$_{improved}$, improves how the file system is checked by utilizing more of the information that resides within the file system image. Table 4 lists the new information-complete, policy-consistent, and secure repairs in SQCK$_{improved}$.

The first three repairs utilize the replicas that ext2 keeps of the group descriptor blocks on disk. While e2fsck does examine these replicas if the primary copy is obviously corrupted, e2fsck misses opportunities to use correct replicas when the primary "looks" fine. Thus, SQCK$_{improved}$ always examines all replicas and performs majority voting across them to determine the correct values; this voting is performed for three important fields: the pointer to the data block bitmap, the inode bitmap, and the inode table. With these fixes, SQCK$_{improved}$ performs information-complete repairs when the pointer to the inode table is corrupted, as desired (described in Section 2). These new repairs are straightforward to implement, requiring only 22 lines of SQL and 40 lines of C.

The fourth repair utilizes the extra information kept in directory ".." fields to repair corrupted directories. First, we fix the false parenthood problem exhibited by e2fsck. With SQCK, we replace the incorrect check of e2fsck originally shown in Figure 7 with the one in Figure 11. This new query elegantly expresses relatively complex

```
SELECT   X.ino, X.start, X.end,
         V.start, V.end,
          (ABS(X.pBlk-V.start)) as distance
FROM     ExtentTable C,
         (A query that returns the start and
          end of a shared extent) AS V
WHERE    X.start <= V.start AND V.end <= X.end
ORDER BY  V.start, distance
```

Figure 12: **Locality-aware repair.** *The query above returns shared blocks that are sorted based on the locality distance from the pointers. The inner query (not shown), stored in Table V, returns a the list of duplicate blocks. The* ABS *command helps sorting the result based on locality distance.*

behavior: it only returns false directory entries in which the child directory does not claim them as a parent with ".."; thus, this false directory entry is correctly cleared instead of that of the rightful parent.

We can extend this repair slightly to write the fifth repair, which corrects even more complicated corruptions of the directory hierarchy. For example, if a path `/a/b/c/` exists and `b`'s inode is corrupted such that `b` no longer appears to be a directory, e2fsck does not do any reconstruction and simply moves `c` to lost+found. However, SQCK$_{improved}$ completely reconstructs the contents of `b` from the back pointers of its children. The complete rule requires a total of 20 new SQL lines with C code similar to that shown in Figure 9.

The sixth repair corrects the allocation policy of e2fsck. Specifically, e2fsck clones data blocks without checking which file is closer to the shared data block. Ideally, the repair should give the existing block to the closest inode and allocate the new clone to the other inode. With SQCK, locality optimizations are easily performed. For example, Figure 12 shows how we utilize the ABS and ORDER BY SQL commands to calculate the distance between a block and its pointer. The bold text shows that the results are sorted on the start of the shared extents and then on the distance between the shared extent and the blocks that point to the extent (X.pBlk). Given this list, SQCK can easily perform the repair such that the shared extent is kept with its closest pointer.

Finally, the seventh repair adds secure cloning. This is done in two ways. First, suppose a corrupt direct pointer incorrectly points to a bitmap block; since the bitmap block is pointed to by more than one group descriptor replica, it is more likely the direct pointer is mistaken than all of the group descriptor replicas; therefore, cloning of that block simply leaks information and does not need to be performed.

```
first_block =  sb->s_first_data_block;
last_block = first_block + blocks_per_group;

for (i = 0, gd=fs->group_desc;
     i < fs->group_desc_count;
     i++, gd++) {
  if (i == fs->group_desc_count - 1)
    last_block = sb->s_blocks_count;
  if ((gd->bg_block_bitmap < first_block) ||
      (gd->bg_block_bitmap >= last_block)) {
      px.blk = gd->bg_block_bitmap;
      if (fix_problem(PR_0_BB_NOT_GROUP, ...))
        gd->bg_block_bitmap = 0;
  }
  ...
}
```

Figure 13: **C-version of Figure 3.** *The C fragment above shows the e2fsck's implementation of the "check block bitmap not in group" shown in Figure 3.*

```
if ((dot_state > 1) &&
    (ext2fs_test_inode_bitmap
     (ctx->inode_dir_map, dirent->inode))) {

  // ext2fs_get_dir_info is 20 lines long
  subdir = e2fsck_get_dir_info(dirent->inode);
  ...
  if (subdir->parent) {
    if (fix_problem(PR_2_LINK_DIR, ...)) {
      dirent->inode = 0;
      goto next;
    }
  } else {
    subdir->parent = ino;
  }
}
```

Figure 14: **C-version of Figure 7.** *The C fragment above shows the e2fsck's implementation of the "check multiple parents" shown in Figure 7.*

Second, suppose a data block is shared by two inodes, one in the `/root` directory and one in the `/home/UserA` directory. In this case, if we want to prevent leaking of information, we might not want to clone the shared block, instead we remove the pointer from the user and keep the one from the root inode. In addition to the existing block conflict check and cloning primitives, this new rule only requires additional two SQL files, for a total of 8 lines to do the path traversal, and 41 lines of C code.

The secure clone repair could be seen as an example where an administrator's decision is more appropriate than an automated one. SQCK does not throw away the need to ask the administrator for the right decision. In such cases, different policies should be present for the administrator to choose from. In SQCK, we can execute different policies easily; each policy is simply mapped to a query or a set of queries.

3.4 Summary

In summary, we have found that declarative queries can succinctly express the many different types of checks and

Improving Scan Time	
1	Reduce seek time with sorted job queue
Improving Load Time	
2	Make the table content compact
3	Only load checked information
4	Use threads to exploit idle time
Improving Check Time	
5	Write queries that leverage indices
6	Leverage fs domain specific knowledge
7	Use bitmaps to reduce search space

Table 5: **Optimization Principles.** *The table lists the optimizations that we have performed such that performance of SQCK is competitive with e2fsck.*

repairs that fsck performs. Our experience also shows that writing checks and repairs in declarative queries is relatively straightforward; each query is written in a few iterative refinement. A complex check or repair, with a little bit of help from C code, can be broken into several short queries that are easy to understand. On average, each query we have written is 7 lines long, and the longest one is 22 lines. Furthermore, only 24 repairs require help from C code. The functionalities of the corresponding C code are generally simple; C code is only used to run a set of queries and iterate the query results. Note that this is different than how C code is used for cross-checking in e2fsck, as illustrated in Figure 13 and 14. Both of the code segments illustrate that a low-level C implementation tends to make a simple check hard to understand and debug. Given such examples, adding the new repairs described in the previous section into e2fsck is likely to complicate the code more.

4 Implementation

We now describe our implementation of the SQCK phases for scanning the file system image from disk, loading the database tables, checking and repairing the structures, and finally flushing the repairs to disk. Our current implementation of SQCK runs on top of a MySQL database and targets the ext2 file system in Linux 2.6.12. When describing our implementation, we focus on the optimizations we found were necessary for achieving respectable performance; Table 5 summarizes these optimizations across the phases.

4.1 Scanning and Loading

In our current implementation, SQCK combines its scanning and loading phases. Conceptually, SQCK maintains a queue of the structures that must be read from disk, processed, and loaded into the tables. As structures are processed, SQCK follows their pointers to determine the next structures. For example, the queue is initialized from the primary superblock; after the superblock, the locations of the group descriptor copies are known; subsequently, the inode tables are processed, which leads to individual inodes and their data blocks.

SQCK implements a number of performance optimizations for scanning and loading. First, to reduce the scan time, SQCK sorts the requests in the queue based on their on-disk locations; sorting the requests minimizes disk head positioning time, especially for file systems that are fragmented. We note that although e2fsck performs a partial optimization of this sort (*i.e.*, directory blocks are sorted before read from the disk [10]), e2fsck is not able to perform the same optimization (*e.g.*, indirect blocks still have to be traversed logically) because it heavily intermixes scanning with checking [19]. SQCK is able to optimize scanning because reading from disk is completely decoupled from checking; hence, SQCK does not need to follow structures in a logical manner.

The primary reason we decouple scanning from checking is because we want to make the common case fast; if corruption is a rare case than our approach improves the overall fsck time. However, there is a tradeoff: if corruption is huge, extra work is needed to invalidate the garbage loaded into the database. Our design is not limited to that only approach; if desired, SQCK can be redesigned by intermixing some phases of scanning and checking according to the structural logical hierarchy. For example, when loading and checking indirect blocks, triple indirect blocks will be loaded and repaired in the database, then only valid double indirect blocks will be loaded to the database, and so on.

Second, SQCK improves load time (and check time) by reducing the size of the initial database tables. Our initial implementation loaded the ext2 structures to match their on-disk format; specifically, SQCK loaded each on-disk pointer as a direct pointer. However, we found that this approach made checking even 100 GB file systems unattractive. Therefore, our next optimization modified the tables to instead use extents to represent pointers referring to contiguous blocks.

Third, SQCK reduces loading time by only loading allocated meta-data. Given that most file systems are half-full [5], a great deal of the inodes are not actually used. To reduce the size of the tables, SQCK does not load the unused inodes into the database tables (though it of course still scans them from disk). However, e2fsck performs one check on unused inodes that SQCK must be able to replicate: e2fsck verifies that each inode with a link count of zero also has a deletion time of zero. To handle this repair, SQCK performs this one check during processing. If SQCK finds a non-conforming inode, that inode is loaded into the table on the fly; to mark that the inode has been repaired, its *used* field is cleared and the *dirty* field is set. We note that this optimization is consistent with the direction in which future file systems are going: ext4 explicitly marks unallocated sections of the inode table to help e2fsck run more efficiently [4].

Fourth, the scanner-loader in SQCK is multi-threaded. Each thread within the pool is able to independently grab a structure from the queue, read the data from disk, process it, and load the information into the corresponding table. Multiple threads allow SQCK to overlap reading requests from disk with loading the tables. As we will see in our evaluation, this optimization is especially important for large partitions.

4.2 Checker

After all metadata has been uploaded into the database tables, SQCK initiates the checking phase, which runs the queries as discussed in the previous section. One important note is that since the checker runs only after the loader, corrupt data can be loaded into the tables. Hence, SQCK provides primitives to invalidate a structure along with the information that originates from it. For example, if the block number that points to an inode table is corrupt, the wrong inodes and the wrong data pointers will be loaded into the table. Later, when the checker discovers that the inode table pointer is corrupt, it simply calls the SQCK primitives to invalidate the corresponding inodes, extents, and directory entries.

The checker has been optimized for performance in three main ways. First, we have found that SQCK must contain appropriate indices for each table; without indices a full scan must be done for each check and joining multiple tables requires a very long time. Thus, each table contains indices over the fields that are checked with the comparison operators.

Given the indices, some queries must be rewritten to leverage them. In our experience, MySQL is not able to always extract the implicit index comparisons in some queries. For example, the check that no directory entry points to an unused inode was originally written as shown in the top half of Figure 15. When the rule was rewritten to make the index comparison explicit, as shown in the bottom part of the figure, the query time improved significantly. Thus, making index comparison explicit is an important principle to do fast checking. We rewrote a total of four queries in this manner, reducing the check time for those four queries from 72 seconds down to just 0.09 seconds on a 1 GB partition.

Second, we have found it beneficial to incorporate file system domain knowledge into the queries. One example is the rule that counts how many blocks are being used in a group. Since SQCK uses extents, it must first select the extents in that group. The naive range-checking query could be written as follows: `(G.start <= X.start AND X.end <= G.end)`. However, given that we know valid extents cannot overlap group boundaries (this has been verified in previous queries), the range-check query can be simplified to `(G.start <= X.start <= G.end)`. This simplified query improves check performance.

```
// find an entryIno that is in the list of
// unused inodes
SELECT *
FROM   DirEntryTable
WHERE  entryIno IN
         (SELECT ino
          FROM   InodeTable
          WHERE  used = 0)
              ---- vs. ----
// find an entryIno that exists in the
// InodeTable and the used field is zero
SELECT *
FROM   DirEntryTable
WHERE  EXISTS
         (SELECT *
          FROM   InodeTable AS I
          WHERE  I.ino  = D.entryIno AND
                 I.used = 0)
```

Figure 15: **Explicit index comparison.** *We rewrite the code to unearth the index comparison.*

The final optimization addresses how to join tables where an index comparison is not possible. For example, the query finding shared blocks across files joins the ExtentTable with itself to find any overlapping extents. We optimize this query by making the search space smaller with bitmaps. For this example, SQCK uses two bitmaps: one for marking used blocks and one for marking shared blocks; the latter bitmap provides a hint as to which extents have overlapping blocks. To find out which part of an extent is actually overlapping, SQCK joins the resulting small table with itself.

4.3 Flusher

Finally, SQCK needs to update any repaired structures to the disk. SQCK is able to determine which structures have been modified by selecting those entries where the dirty flag is set. Following the same behavior as e2fsck, SQCK updates the structures in-place on disk (*i.e.*, it does not currently use a separate journal).

To ensure the metadata writes are ordered correctly [16], currently SQCK performs a series of queries ordered by the dependency graph; the graph ensures that blocks are updated before the pointers to those blocks. In the next generation of SQCK, a journaling facility will be added to ensure that a crashed repair process will not modify the old data partially.

	C Code		**SQL**
Component	**LOC**	**; count**	**LOC**
Scanner	2759	1378	–
Loader	609	177	103
Checker+Repair	***2527**	1468	910
Primitives	695	348	98
Flusher	114	49	27
Total	6704	3420	1138

Table 6: **SQCK$_{improved}$ LOC.** *The table presents the complexity of SQCK$_{improved}$. Scanner includes threads and functions that process the structures. (*) The C code for the checkers and repairs are mostly wrappers that call the SQL files.*

	SQCK	**ext2**	**ReiserFS**	**XFS**
LOC	2527	16472	11281	21773
# Chks	121	121	156	344
Instr. gap	16 ± **16**	71 ± **161**	56 ± **203**	128 ± **257**
Func. gap	1 ± **1**	4 ± **6**	1 ± **3**	5 ± **6**
# Chk func.	121	31	32	72
# Chks/chk-func	1 ± **0.1**	4 ± **5**	5 ± **8**	5 ± **5**

Table 7: **Checkers complexity.** *The table shows the logical complexity of SQCK$_{correct}$, ext2, ReiserFS and XFS checker codes (excluding libraries). Standard deviation is shown right next to the ± sign. "Inst. and Func. gaps" quantify the number of C instructions and functions separating one check from the next check. "# Chk func" shows in how many functions the checks are diffused. Finally, "# Chks/func" averages the number of checks performed in each checker function.*

5 Evaluation

In this section we evaluate SQCK along three axes: complexity, robustness, and performance. Overall, we believe many of the goals of SQCK have been achieved.

5.1 Complexity

Table 6 presents the complexity of SQCK$_{improved}$, the most complete version of SQCK. As the table suggests, SQCK is comprised of C and SQL code. The scanner is the only place where the complexity of the C code still exists. However, the code is generally simple because it scans the file system in a logical hierarchy. The checker code looks big, however, it is mostly wrapper functions that call the corresponding queries; most wrappers consist of the same 15 lines of C code. A generic wrapper could be built to reduce the amount of C code.

SQCK so far has been written all at once by one small group. Thus, it is possible that SQCK will become more complex when developed by a bigger group over a longer period of time. However, we believe the core power of SQCK lies within the simple and robust queries; each query consists of 7 lines of code on average. These queries decouple the checks from the C code, enabling us to maintain reliability in an easier way. Compared to e2fsck, which consists of 16 thousand LOC of cluttered checks and repairs and 14 thousand LOC of scan utilities, all written in low-level C code, SQCK can be considered a big step towards simplifying file system checkers.

To show that we are solving a broader significant problem, Table 7 attempts to quantify the logical complexity of ext2, ReiserFS, and XFS checker utilities, all written in C. The metrics shown in the table are generated by our parser written using CIL [25]. In fsck-related code, we annotate the location where each check is performed. The parser computes the complexity-metrics as described in the table. For example, we compute how many instructions and function calls separate each neighboring checks. If the numbers are high, the checks are most likely diffused and reasoning about their correctness might be nontrivial, if not impossible. The numbers reported in Table 7 exclude fsck libraries (*e.g.*, scanner), hence they only depict the logical complexity of the checker component.

We make two important observations: First, the average number of C instructions and functions that separate two checks are high in all fsck utilities, with significant standard deviations; the separation can be as low as 4 or as high as 1700 instructions. Second, checks are greatly diffused in many functions; a function could make a small number of checks while some other could perform as many as 47 checks. In such implementations verifying that all checks are complete and ordered correctly can be cumbersome. On the other hand, SQCK hides the complex logic of the checks in declarative queries, greatly reducing the gap between neighboring checks; the standard deviations shown in the SQCK column illustrate the neat organization we have achieved. In summary, we believe all C-implementations of fsck are likely to suffer from the same problems as e2fsck.

5.2 Robustness

To test the robustness of the three versions of SQCK, we have injected a total of 356 corruption scenarios. In each scenario, we injected corruption into one or more fields to test whether a check and the corresponding repair behave as expected. First, we have injected 55 faults to ensure that SQCK$_{fsck}$ passes most of the interesting repairs out of the total 121 repairs. Injecting faults for the rest of the tests should be straightforward. Second, we have verified that SQCK$_{correct}$ passes the fsck reliability benchmark described in Section 2 by injecting additional 180 faults. Finally, we injected 10 new faults to test the new repairs that we introduced in SQCK$_{improved}$ and verified the resulting repairs.

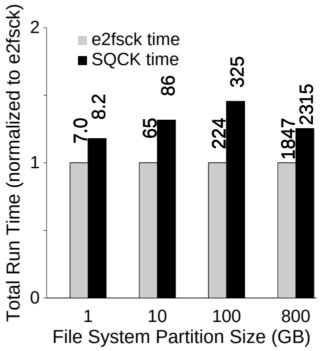


Figure 16: **Overall runtime comparison .** *The bar graph shows the comparison of the total runtime of e2fsck and fully optimized SQCK for different file system sizes. The bars are all normalized to e2fsck runtime and the SQCK bar show the relative slowdown. The absolute runtime figures in seconds are shown on top of the bars.*

5.3 Performance

The experiments in this section were performed on an 2.2 GHz AMD Opteron machine with 1 GB memory and 1 TB WDC WD10EACS disk. We used Linux 2.6.12, e2fsck 1.39, and MySQL 5.0.51a. The tables are mounted on a 512 MB ramdisk.

We test the performance of SQCK and e2fsck on four partitions with different sizes: 1, 10, 100, and 800 GB. Each of the partition is made half-full [5] by filling it with the root file system image of a machine in our laboratory along with a large number small files from kernel builds and large files from virtual machine images.

Figure 16 shows e2fsck compared to our fully optimized SQCK. The fully optimized SQCK incorporates all the principles described in Table 5; specifically, it sorts the block scan, loads extents and linked inodes only, uses 16 worker threads, and uses fast queries. In our first generation prototype we managed to keep the running time of SQCK within 1.5 times of e2fsck runtime.

We show in more detail how each of the scan and load optimization principles improve the runtime significantly by turning off one optimization feature at a time. The runtime of each of these unoptimized versions are compared relative to the fully optimized SQCK.

First, the sorted job queue is disabled such that we scan the file system logically. Figure 17 shows that for a large file system (*e.g.,* 800 GB), sorting the job queue plays a significant role; scanning the file system logically takes almost 3 times as long as the fully optimized one. Note that in this experiment, we disabled the loading phase to compare only the scan performance. The serial scanning for the 100 GB file system is 8 seconds faster than the fully optimized SQCK because the file system was almost not fragmented at all; the advantage of the sorted scanning is noticeable for fragmented and/or big file systems.

Second, we show the importance of making the initial table compact. Figure 18 shows the slowdown of two unoptimized versions: one that loads all inodes, and one that loads direct pointers instead of extents. When loading all inodes, the runtime is increased significantly; for 800 GB file systems, 97 million inodes will be loaded out of which only 900 thousand have non-zero link counts. When loading direct pointers, the runtime increases dramatically. For the 100 GB file system, the DirectPointerTable already consumes 360 MB, while the ExtentTable only consumes 9 MB.

Third, Figure 19 shows how multiple threads enable us to significantly overlap scan and load time. When the number of worker threads is reduced to one, the slowdown is almost 1.5 times in all file systems. For large file systems, increasing the number of threads gives a faster runtime; at 800 GB, using 16 worker threads improves the runtime.

In summary, our evaluation of the first generation prototype of SQCK shows that SQCK obtains comparable performance to e2fsck. In the next generation of SQCK, we plan to perform two additional enhancements. First, some checks can be merged so that the table-scan time can be reduced. If the checks find a problem, then nested sub-checks will be run to pinpoint the actual problem. Second, we plan to run some checks and repairs concurrently by utilizing the information dependency graph in Figure 10. The graph provides the dependency tree that tells which checks and repairs are safe to run in parallel. With a faster overall check time, we hope file system developers will be encouraged to write as many rules as needed.

6 Related Work

We believe that SQCK is the first tool to apply a declarative query language for checking and repairing file systems. We briefly compare SQCK to research on specification and declarative languages and efforts focused on improving fsck.

Specification languages like Alloy [21] and Z [11] are useful for describing constraints of a system and then finding violations of that model. Similar in goals to SQCK, Demsky and Rinard took this approach for automatically repairing file systems [12]; however, we believe the drawback of their work is that it does not allow one to naturally express the *repairs* that should be performed when violations are discovered.

Numerous researchers have recently explored the advantages of using declarative languages in other domains, such as system configuration [13] and network overlays [23].

Finally, other researchers have proposed a technique for optimizing the performance of file system check-

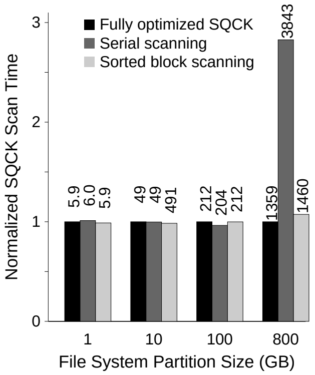


Figure 17: **Scan time improvement with sorted queue.** *This bar graph shows the time to scan each file system without loading. In each set, the left-most to right-most bars show the fully optimized SQCK, the logical scan, and the sorted scan with only 1 thread. The values are normalized to the fully optimized SQCK time.*

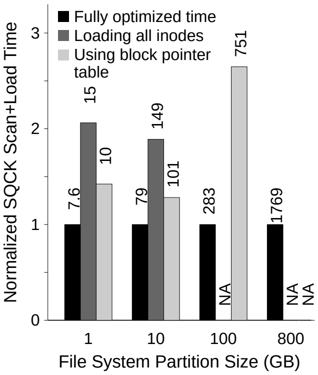


Figure 18: **Making the table compact.** *The bar graph shows the slowdown of scan and load time when we load big tables. In each set, the left-most to right-most bars show the fully optimized SQCK, fully optimized SQCK but with loading all inodes, and loading direct pointers instead of extents. The values are normalized to the fully optimized SQCK time. "NAs" imply experiments that do not finish in 3 hours.*

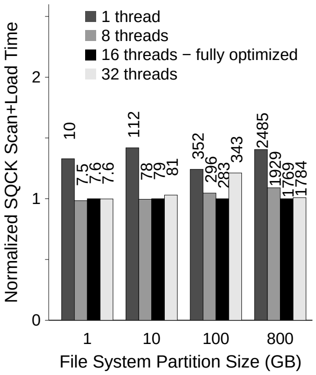


Figure 19: **Overlapping scan and load time.** *The bar graphs show the runtime of different runs that use different number of threads. With one thread, the runtime is the worst as we cannot utilize the idle time during scanning. The values shown are normalized to fully optimized SQCK time with 16 threads.*

ers [20, 26]. The basic approach in both checkers is to track which portions of the disk are being written to and focus repair on only those active portions of the disk. While this technique can certainly improve performance, it does not simplify the implementation of the checker nor enable fundamentally new repairs, as does SQCK.

7 Conclusion

Recovery code is complex and hard to get right. Current approaches describe recovery at a very low-level: thousands of lines of C code. One approach to improving the state of the art is to apply more formal techniques that find bugs in such code [14] and thus evolve the code towards a less-buggy future.

We instead advocate a higher-level strategy. By encapsulating the logic of a file system checker in a set of declarative queries, we provide a more concise description of what the checker should do. Complexity is the enemy of reliability; by paring down the checker to its declarative core, we believe we have taken an important step towards improving the robustness of file system checking.

Acknowledgments

We thank the anonymous reviewers and Greg Ganger (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group, in particular Nitin Agrawal, Sriram Subramanian, and Swaminathan Sundararaman for their insightful comments. Finally, we thank Yupu Zhang who helped us in annotating the XFS fsck.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] http://www.mysql.com/.

[2] http://e2fsprogs.sourceforge.net/.

[3] http://en.wikipedia.org/wiki/SQL.

[4] http://en.wikipedia.org/wiki/ Ext4.

[5] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *FAST '07*, San Jose, CA, February 2007.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*, San Diego, CA, June 2007.

[7] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and

Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, pages 223–238, San Jose, California, February 2008.

[8] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *DSN '08*, Anchorage, Alaska, June 2008.

[9] George Candea and Armando Fox. Crash-Only Software. In *HotOS IX*, Lihue, Hawaii, May 2003.

[10] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[11] Jim Davies and Jim Woodcock. *Using Z: Specification, Refinement, and Proof* . Prentice Hall, 1996.

[12] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA '03*, Anaheim, California, October 2003.

[13] John DeTreville. Making system configuration more declarative. In *HotOS X*, Sante Fe, New Mexico, June 2005.

[14] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *VMCAI '04*, Venice, Italy, January 2004.

[15] Rob Funk. fsck / xfs. http://lwn.net/Articles/226851/.

[16] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

[18] Roedy Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, February 2005.

[19] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, September 2007.

[20] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *HotDep II*, Seattle, Washington, Nov 2006.

[21] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

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

[23] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05*, Brighton, UK, October 2005.

[24] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[25] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *CC '02*, pages 213–228, April 2002.

[26] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *USENIX '98*, pages 77–89, New Orleans, LA, June 1998.

[27] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[28] American Data Recovery. Data loss statistics. http://www.californiadatarecovery.com/content/adr_loss_stat.html.

[29] David M. Smith. The Cost of Lost Data. http://gbr.pepperdine.edu/033/dataloss.html.

[30] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[31] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*, Seattle, WA, November 2006.

[32] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.