Understanding Data Races in MySQL

Wentao Wu, Jiexing Li, Tao Feng, Xiaofeng Zhan
University of Wisconsin-Madison

Abstract

Data races are notorious for their close relationship to many painful concurrency bugs that are very difficult to locate. On the other hand, it is both impossible and unnecessary to forbid every data race in large software systems, by locking every shared variable. It is impossible since we cannot afford for the performance deterioration by locking everything. It is unnecessary since most of the data races are actually benign, i.e., they do not compromise program's correctness.

In this paper, we study the behavior of data races inside MySQL, a modern database management system. We collect a large set of data races reported by Helgrind, after running a benchmark on MySQL that contains SQL query workloads coming from typical database applications. We then carefully examine these races and work out a taxonomy on programming paradigms that can cause data races. To learn the semantics implied in these paradigms, we further study the roles of the shared variables that are involved in the races, based on their read/write histories. Our findings in this paper will be helpful for programmers to get a deeper understanding on how to write more reliable multithreading programs.

1 Introduction

Data races occur when multiple threads access the same memory address without any synchronization mechanism, and at least one of these accesses is a write. Identifying data races will be very helpful for debugging multithreading programs, since they are often closely related to many painful concurrency bugs, which are notoriously difficult to locate, especially in large software systems.

A straightforward solution that eliminates all possible data races is to lock every shared variable. However, this solution may dramatically degrade the performance of the system. First, locking itself is an expensive operation. Large software like database systems usually involve hundreds of thousands of shared memory units, hence locking all of them is actually impossible in practice. Second, using locking heavily will also increase the chance of deadlock inside the system. Therefore, for performance concern, data races cannot be completely ruled out from modern software systems.

Since data races may cause unexpected runtime errors and cannot be entirely inhibited, understanding their behavior inside the system becomes an important issue. According to some recent study [7], most of the data races are actually *benign*, i.e., they do not compromise program's correctness. The authors in [7] further give five reasons for benign races: *User Constructed Synchronization*, *Double Checks*, *Both Values Valid*, *Redundant Writes*, and *Disjoint Bit Manipulation*. Each of these reasons has typical programming paradigms at source code level. For example, a typical program snippet for a double check is:

```
if (x) { lock (...) { if (x) ... } }
```

Here, the read in the first check is not protected by any lock, so there can be a data race on it if other threads try to modify `x` at the same time. However, this does not affect the correctness since the value of `x` will be checked again after obtaining the lock, and therefore the races on the first `x` is benign.

We believe that understanding these programming paradigms that cause data races will be very helpful for programmers to identify bugs and write more reliable code when developing multithreading programs. For example, double checks can be safely used since we know that they will only lead to benign races. The inspiration here is the same as the idea behind *design patterns* [3], which is one of the most influential achievement in the history of software engineering. The job of coding a large system will be much easier by mastering a set of reusable patterns that have been demonstrated to work well by many previous systems.

Unfortunately, the discussion in [7] on these programming paradigms is only superficial, without giving any

more example patterns except for the one on double check, as described above. Meanwhile, to the best of our knowledge, we are also not aware of any previous work that tries to develop such a set of programming paradigms, which we think will be quite useful.

In this paper, we report our effort towards building a set of common programming paradigms that can cause data races. We systematically study the data races occurring in a mature software called MySQL[1], and then work out a taxonomy of programming paradigms that will cause data races. MySQL is an open-source database management system that has been developed for more than 15 years. We choose MySQL as our target system due to the following four reasons:

- Database systems are typical large software in which concurrency mechanisms are heavily used to gain high performance and throughput.
- Unlike another well-known open-source database system PostgreSQL[2], pthreads[3] are intensively used inside MySQL, which makes race detection tools like Helgrind[4] workable.
- MySQL is sufficiently large and complex for covering a complete set of common programming paradigms, which can be thought of as ubiquitous in other multithreading programs.
- Since MySQL has been developed for many years, there should be very few bugs, and most of the races are expected to be benign. Therefore we have a great chance to learn which programming paradigms are *good*.

The rest of the paper is organized as follows. Section 2 provides background information on MySQL and Helgrind. Section 3 describes our methods on obtaining and analyzing data races in MySQL. Section 4 presents our results, with focus on the taxonomy of programming paradigms we found through our analysis to the data races. To better understand the semantics implied by these paradigms, in Section 5, we further study the roles of those shared variables that are involved in the races. We discuss related work in Section 6, and conclude the paper in Section 7.

2 Background

In this section, we briefly introduce some background information about MySQL and Helgrind.

[1] http://www.mysql.com/
[2] http://www.postgresql.org/
[3] http://en.wikipedia.org/wiki/POSIX_Threads
[4] http://valgrind.org/docs/manual/hg-manual.html

2.1 MySQL

MySQL is an open-source database management system. Figure 1[5] shows the architecture of MySQL. Basically, MySQL server uses a *layered* architecture. Clients send SQL queries to the server via standard interface such as ODBC and JDBC. A SQL query will first be parsed and then optimized to generate a *query plan*, which is a tree-like structure with each node representing a relational operator. The plan will then be executed by the *executor*, which simply runs each operator by traversing the tree in a bottom-up manner.

The most important module of MySQL server is its storage engine. Starting from 5.5, the version we use, MySQL adopts *InnoDB* as its default storage engine, which provides standard ACID-compliant transaction features. Prior to MySQL 5.5, *MyISAM* is used by default, which does not support transactions. The storage engine manages all the important resources in a database that will be accessed by multiple queries, including buffer pages, indexes, locks, logs, and files. Not surprisingly, this is the module where most of the data races are involved.

2.2 Helgrind

Helgrind is a tool included in the *Valgrind*[6] toolkit that can detect data races when running a binary program. The latest version (3.6.1) we use adopts an race detection algorithm based on Lamport's *happens-before* relation [5]. Simply speaking, it defines an ordering on a set of asynchronous events, and reports races on the cases when this ordering is violated. Compared with *lock-set* [9] based algorithms, happens-before algorithms can report fewer false positives and hence can be more accurate, although they will also miss certain kinds of races.

To implement the happens-before algorithm, Helgrind needs to intercept each call to the events it concerns. However, it can only intercept standard pthread calls. As a result, Helgrind is not aware of any user constructed synchronization mechanism, and will report a race in this situation that will not actually happen.

3 Method

Our goal is to capture a large set of data races that can appear when running MySQL. To achieve this, we design a benchmark consisting of different kinds of SQL query workloads. We admit here that there is no way for us to know whether the set of races obtained is *complete*. However, we take care in designing our benchmark so

[5] From http://dev.mysql.com/doc/refman/5.1/en/pluggable-storage-overview.html
[6] http://www.valgrind.org/

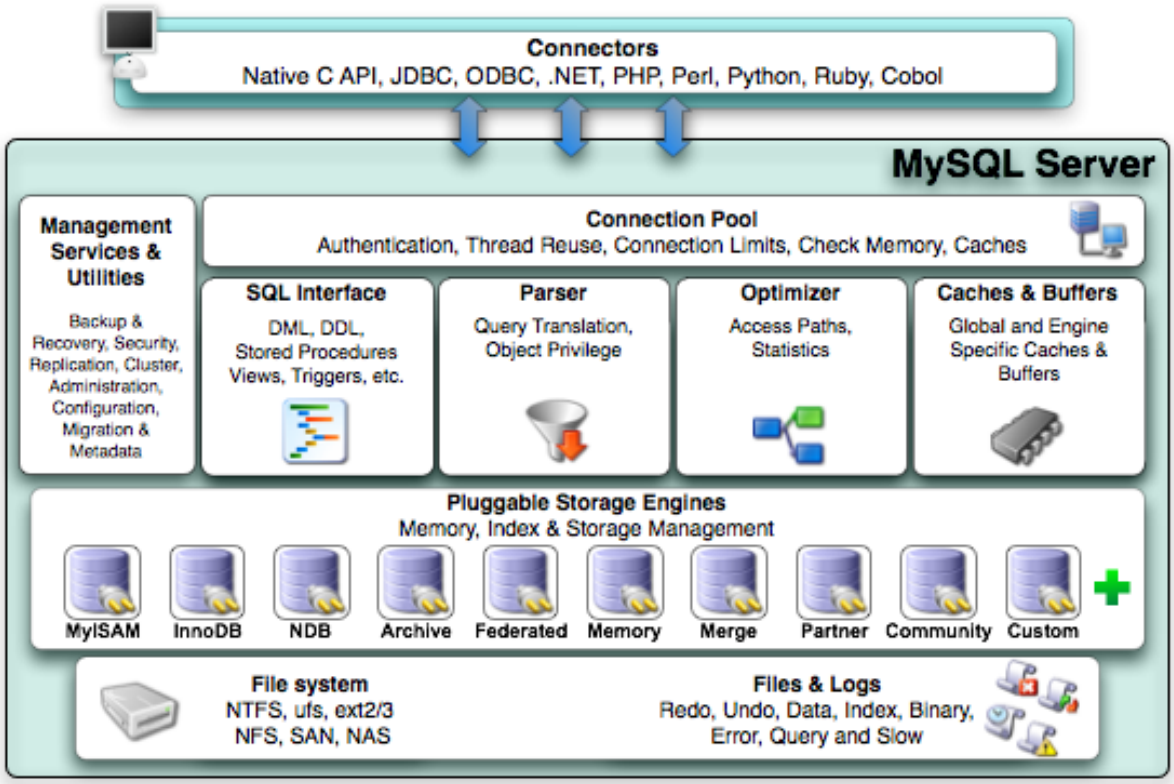


Figure 1: Architecture of MySQL

that it contains typical workloads that can raise in the daily use of a database. Therefore we believe that the data races triggered by this benchmark should contain most of, if not all, typical data races inside the database system.

In this section, we first present our benchmark queries in Section 3.1. Then, in Section 3.2, we outline the steps of our method.

3.1 Benchmark Queries

We create two databases with respect to the schemas specified in the TPC-H[7] and TPC-C[8] benchmarks, respectively[9]. We then load 1GB data into each database, by using the *dbgen* tool that is freely available on the TPC website.

We design a benchmark including four different workloads:

- **W1**. The first workload consists a set of simple read/write queries that are randomly generated based on the TPC-H schema, which only update *tables*. A simple read query is a `SELECT` statement that involves a single table, while a simple write query is an `UPDATE`, `INSERT`, or `DELETE` statement that involves a single table.
- **W2**. The second workload consists a set of simple read/write queries that are randomly generated based on the TPC-H schema, which update both *tables* and *indexes*.
- **W3**. The third workload consists a set of TPC-H queries. TPC-H benchmark contains typical online analytical processing (OLAP) queries which are *read-intensive*, involving complex joins across multiple tables.
- **W4**. The fourth workload consists a set of TPC-C queries. TPC-C benchmark contains typical online transaction processing (OLTP) queries which are *write-intensive*.

Since access to the data in the database is either via *file scan* or *index*, workloads W1 and W2 are expected to cover typical access paths inside the database.

[7] http://www.tpc.org/tpch/
[8] http://www.tpc.org/tpcc/
[9] See the TPC-H and TPC-C specifications that can be freely downloaded from their websites for the description of database schemas.

Meanwhile, workloads W3 and W4 come from standard benchmarks that emulate typical workloads that can occur in real application scenarios, and they can cover most of the races that will actually raise during the normal execution of the database server.

3.2 Steps

Our study on data races then proceed in the following three steps:

1. We use Helgrind to run the MySQL server, which can record the data races during the execution of the server.
2. We run each workload respectively:
   - For W1 and W2, we concurrently run two clients, with each sending 10 SQL queries to the server. We also vary the percentage of read queries in each workload and rerun the clients to try to prob new data races.
   - For W3, we pick up 3 queries (see Appendix A) from the 22 queries available in the standard TPC-H benchmark. We then concurrently run two clients, with each sending these 3 queries to the server.
   - For W4, we use a tool *dbt2*[10] that can automatically generate TPC-C queries and run them on top of MySQL. We slightly modify the tool so that MySQL server could be run under the instrumentation of Helgrind. The command we used to run the tool is:

     ```
     run_workload.sh -c 2 -d 1200 -w 2
     ```

     , which means we run 2 clients for 1200 seconds on 2 warehouses.
3. We collect all the data races recorded by Helgrind, and manually analyze and classify them into different categories.

As noted above, we use relatively *lightweight* instances for each workload. The reason is twofold. First, running MySQL under Helgrind is much slower, and we cannot afford heavier workloads with the computer[11] we used in our test. Second, we are only interested in data races that exhibit *different programming paradigms*. Running heavier workloads may raise more data races, but whether they can come up with more *types* of programming paradigms is unclear. We do not believe there will be many, since programming paradigms are directly related to the semantics implied by the programmer. It is not very convincing that the programmer will use some code pattern that can only cause data races on heavy workloads but never cause data races on low workloads. Nonetheless, it is still interesting to investigate the difference when heavier workloads with more clients and more queries are issued, by using a more powerful computer.

[10] http://sourceforge.net/projects/osdldbt/files/dbt2/

[11] We use a laptop configured with 2.1 GHz Intel Core Dual CPU and 2GB memory.

4 Results

We present the results of our study in this Section. Section 4.1 summarizes the distribution of data races across different modules inside MySQL server. Section 4.2 gives a taxonomy on the programming paradigms that can cause data races, based on the races we collect.

4.1 General Statistics

Figure 2 shows the distribution of data races on workloads W1 and W2. We measure the numbers by varying the percentage of read queries in 0%, 25%, 50%, 75%, and 100%, respectively. We also measure the number of data races when no query is issued to the database (denoted by '*No query*' in the plot).

We observe three interesting phenomena:

- The distribution is quite uneven. Modules such as buffer manager ('/storage/innobase/buf'), file manager ('/storage/innobase/fil'), lock manager ('/storage/innobase/lock'), and log manager ('/storage/innobase/log') include most of the data races detected since they are the key modules that manage shared resources inside the storage engine.
- The peak number of races does not occur when there are pure read or pure write queries, but occurs at a certain mixture of read and write queries. In our test, the maximum number of races (1387) on W1 occurs when there are 25% of read queries, while the maximum number of races (1552) on W2 occurs when there are 50% of read queries.
- W2 raises more data races than W1, since it involves more index updates. This can be confirmed by the slight increase on the number of races in the index module ('storage/innobase/btr'). Since we only use lightweight instances of the workloads, the difference is not very significant. We can expect more races on indexes by using heavier workloads.

Figure 3 further shows the distribution of data races when running workloads W3 and W4. Since W3 is read-intensive, we also compare the distribution with that in W1 and W2 when there are pure reads. As can be seen,

W3 raises much more races overall. However, most of the new data races appear in the modules of buffer manager ('/storage/innobase/buf') and synchronization control ('/storage/innobase/sync'). This is expected since TPC-H queries involve heavy table scans that incur intensive competition on using the buffer. For W4, since it is write-intensive, we compare its distribution with that in W1 and W2 when there are pure writes, and we can observe similar variations as well. Apart from the huge number of races in the buffer manager and synchronization control modules, there are also considerable number of races in the log management module ('/storage/innobase/log'), which are not observable under read-intensive environment since only writes to the database should be recorded for recovery purpose on system crash.

4.2 Taxonomy of Programming Paradigms

As discussed in Section 1, previous study in [7] reveals that benign data races are related to different programming paradigms. We carefully examine the data races reported in MySQL, and present our findings in this section. In summary, we identify four typical programming paradigms: *User Constructed Synchronization*, *Disjoint Bit Manipulation*, *Repeated Checks*, and *Approximated Values*. The first two are already described in [7], while the last two can roughly fall into the category of *Both Values are Valid*, also mentioned in [7]. However, we feel that the category of *Both Values are Valid* is both too big and too vague, so we want to break it into finer subcategories. There are two other categories in [7] that we have not found their way in MySQL: *Redundant Writes* and *Double Checks*. *Redundant Writes* is admitted by the authors of [7] as another subcategory of *Both Values are Valid*, while *Double Checks* has been illustrated in Section 1. To some extent, we think that the absence of these two categories makes sense, since these two paradigms are actually weird and should not appear in a well-formed program. For example, the first check in the example of *Double Checks* shown in Section 1 seems totally redundant.

In the following discussion, for each paradigm, we will first illustrate it with pseudo code, and then present several case studies to show how it is used inside MySQL.

4.2.1 User Constructed Synchronization

For portability reasons, except for using standard pthread functions, large multithreading systems like MySQL will also implement its own functions for synchronization purpose. These synchronization mechanisms are unaware of by race detection tools like Helgrind that are based on intercepting pthread calls. Therefore, a number of races reported are due to user constructed synchronization mechanisms. Strictly speaking, the races reported here should be categorized as false positives, since there are actually *no* races. However, to be familiar with this kind of programming paradigms can help programmers identify these *fake* races more quickly.

The pseudo code description looks like:

```
Thread 1:
    mutex_enter(&mutex);
        update X;
    mutex_exit(&mutex);

Thread 2:
    mutex_enter(&mutex);
        update X or read X;
    mutex_exit(&mutex);
```

There is actually no race here, since every access to the `X` is protected by user defined synchronization primitives `mutex_enter()` and `mutex_exit()`. Unfortunately, Helgrind cannot recognize these functions for synchronization purpose, and will report data races on `X`.

Case Study[12]:

```
Thread 1:
fil_flush_file_spaces(){
   mutex_enter(&fil_system->mutex);
   n_space_ids = UT_LIST_GET_LEN(
     fil_system->unflushed_spaces);
   mutex_exit(&fil_system->mutex);
}

Thread 2:
fil_flush(){
   mutex_enter(&fil_system->mutex);
   UT_LIST_REMOVE(
      unflushed_spaces,
      fil_system->unflushed_spaces,
      space);
   mutex_exit(&fil_system->mutex);
}
```

In this code snippet, both threads access the variable `fil_system->unflushed_spaces`, which is protected by `mutex_enter` and `mutex_exit`. However, data races are still reported by Helgrind. Similar cases appear in MySQL for many times. Interestingly, there are even data races reported inside `mutex_enter()` and `mutex_exit()`. See Appendix B for a detailed study on the implementation of these two functions.

[12] See Appendix D for the locations where the functions are defined.

4.2.2 Disjoint Bit Manipulation

This type of programming paradigms is related to data races reported on a shared variable that is a complex structure with multiple fields. Different threads involved in the races usually operate on different fields inside the structure. The pseudo code description looks like:

```
typedef struct { X; Y; } A;

A a;

Thread 1:
    write a->X;

Thread 2:
    read or write a->Y;
```

However, data races are reported by Helgrind on this kind of programming paradigms, since Helgrind regards the structure `A` as an atomic unit. In other words, access to any field of `A` is treated by Helgrind as equivalent to access to `A` as a whole. Clearly, such data races are sheerly benign.

Case Study:

```
Thread 1:
srv_release_threads(){
   ...
   slot->suspended = FALSE;
   ...
}

Thread 2:
srv_thread_has_reserved_slot(){
   mutex_enter(&kernel_mutex);
   if(slot->in_use &&
            slot->type == type){
      occupy the reserved slot;
   }
   mutex_exit(&kernel_mutex);
}
```

In the above example, `slot->suspended` is written by Thread 1 and `slot->in_use` and `slot->type` are read by Thread 2, which raises a data race reported by Helgrind. If Thread 1 is preempted before setting `slot->suspended` as `FALSE`, Thread 2 is not affected since it does not need to refer to `slot->suspended` during its execution. Note that `&kernel_mutex` protection is used in Thread 2 to prevent multiple threads occupying the same slot.

4.2.3 Repeated Checks

The pseudo code description looks like:

```
Thread 1:
    Loop:
    if (condition on X) {
        do something;
    } else {
        do something else;
        goto Loop;
    }

Thread 2:
    X = new_value;
```

Clearly, there is a race here. In Thread 1, variable `X` is read and the condition containing `X` is evaluated. Thread 2, at the same time, is going to update `X`. Suppose the condition can only be triggered after `X` is assigned the new value. The race here then will not affect the correctness of the program, since Thread 1 will keep on checking the condition until it becomes valid. The only side effect is that Thread 1 may waste some time on checking `X` again if Thread 2 is preempted by Thread 1 before `X` is updated.

Case Study 1:

```
Thread 1:
buf_pool_check_no_pending_io(void){
   bool ret = TRUE;
   if (buf_pool->n_pend_reads) {
      ret = FALSE;
   }
   return ret;
}

Thread 2:
buf_page_io_complete(buf_page_t*
        bpage){
   ...
   processing read requests
   buf_pool->n_pend_reads--;
   ...
}
```

In the above code snippet, Thread 1 checks whether there is pending read requests while Thread 2 will process them if any. Consider the following case. In Thread 2, `buf_pool->n_pend_reads` is initialized to be 1 and will become 0 after the decrement. Now, suppose that Thread 2 is preempted by Thread 1 before the decrement is done. In this situation, Thread 1 will return `TRUE`, indicating that there are still pending read

requests. The consequence (not shown here) is that, MySQL will try to process pending read requests in the buffer again, resulting in finding that there is actually no such requests. Although this behavior leads to wasting time on unnecessary checks, the correctness is not affected.

Case Study 2:

```
Thread 1:
logs_empty_and_mark_files_at_shutdown(
                      void){
   ...
   srv_shutdown_state
            = SRV_SHUTDOWN_CLEANUP;
   ...
}

Thread 2:
srv_lock_timeout_thread(){
   loop:
   ...
   if (srv_shutdown_state
            >= SRV_SHUTDOWN_CLEANUP){
      goto exit_func;
   }
   goto loop;
   exit_func:
   ...
}
```

As suggested by the function names, Thread 1 marks the state of server to be `SHUTDOWN`, and Thread 2 checks whether a thread has exceeded the timeout when waiting for a lock. If Thread 1 is executed before Thread 2, then Thread 2 will execute `exit_func`. However, if Thread 1 is preempted by Thread 2 before setting the `srv_shutdown_state`, then Thread 2 will not execute `exit_func` immediately. Instead, it keeps on checking the value of `srv_shutdown_sate`, until Thread 1 successfully does the assignment. In this case, the execution of `exit_func` is delayed. However, it does not affect the correctness since `exit_func` will finally be executed.

4.2.4 Approximated Values

Not every variable requires an *accurate* value all the time during the execution of the program. For example, large database systems like MySQL keep many variables that are only used for *statistical* purpose. There are many different usages of these statistics. Some are just runtime parameters indicating certain system states such as the number of reads/writes on buffer pages, which are only used for monitoring and administrative purpose. Some others may be used by higher-level modules such as the query optimizer. No matter which purpose, the values recorded in these variables do not need to be perfectly precise, and hence it is not worthwhile to use expensive synchronization methods to protect them.

The pseudo code description looks like:

```
Thread 1:
    update X;

Thread 2:
    read or print X;
```

The key difference here from the case of *Repeated Checks* is that the value of X will never be used to determine the *control flow* of the program.

Case Study 1:

```
Thread 1:
btr_cur_search_to_nth_level(){
   ...
   btr_cur_n_non_sea++;
   ...
}

Thread 2:
srv_refresh_innodb_monitor_stats(void){
   ...
   btr_cur_n_non_sea_old
            = btr_cur_n_non_sea;
   ...
}
```

In this example, Thread 1 searches a B-tree index and positions a tree cursor at a given level. The variable `btr_cur_n_non_sea` is used to record the number of non-hashing searches down the B-tree. Thread 2 stores the current statistic item to another location. Both variables in Thread 2 are used to calculate per-second average statistics. The statistics will be displayed on the monitor for users' reference and will not be involved in any critical functionality of MySQL.

Case Study 2:

```
Thread 1:
os_file_read_func(ulint n){
   ...
   os_bytes_read_since_printout += n;
   ...
}

Thread 2:
os_aio_refresh_stats(void){
```

```
    ...
    os_bytes_read_since_printout = 0;
    ...
}
```

In this example, the variable in Thread 1 records the number of bytes that are read since the last printout, and Thread 2 resets the statistic to be 0. The function `os_aio_refresh_stats(void)` is called in another function periodically, protected by the mutex `&srv_innodb_monitor_mutex`. This means that, in the above case, Thread 2 can preempt Thread 1 but Thread 1 cannot preempt Thread 2. The consequence of this preemption is that Thread 1 may not count the most recent number of the bytes read. Since the statistic is not critical, the deviation can be ignored.

Case Study 3:

```
Thread 1:
trx_assign_rseg(){
    rseg = trx_sys->latest_rseg;
    ...
}

Thread 2:
trx_assign_rseg(){
    ...
    rseg = UT_LIST_GET_NEXT(
              rseg_list, rseg);
    ...
    trx_sys->latest_rseg = rseg;
}
```

In this example, the first and last statement in the same function cause a data race. The function assigns a rollback segment to a transaction in a round-robin fashion. All the rollback segments for a transaction are kept in a linked list. The first statement loads the latest rollback segment to `rseg`, then it goes to the segment list to search for the next segment following the current one and update `rseg`. The last statement assigns `rseg` to the latest rollback segment. If Thread 2 is preempted before completing the last statement, the latest rollback segment will not be the latest one, but an older version. However, the rollback segment is still updated in a round-robin fashion, and the only effect from this race is that the interleaving order on threads may be slightly changed. Nonetheless, this will not affect the correctness of the program.

5 Role of Shared Variables

The study presented in Section 4.2 mainly focuses more on the *syntactic* level of the programming paradigms. To further understand the *semantics* implied by these paradigms, we investigate the shared variables that are involved in the reported races. In this paper, we only focus on classifying these shared variables based on their read/write histories. Basically, any shared variable with data races on it can fall into one of the following five categories:

- *Write-Only*: the variable is only written but never read.
- *Single-Reader-Single-Writer*: the variable is read by one thread and written by another thread.
- *Single-Reader-Multiple-Writer*: the variable is read by only one thread but written by multiple other threads.
- *Multiple-Reader-Single-Writer*: the variable is read by multiple threads but written by only one thread.
- *Multiple-Reader-Multiple-Writer*: the variable is both read and written by multiple threads.

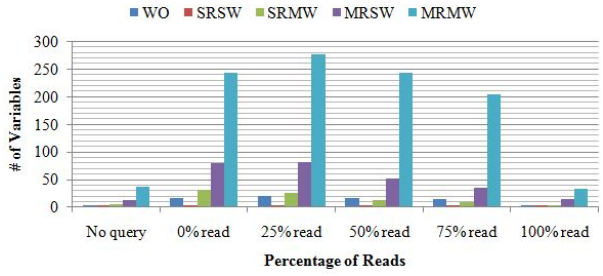
Different access histories imply different functionalities of the variable. A write-only variable sounds weird, since its value will never be read and therefore the data races on it can be totally ignored. Theoretically, it seems that such a variable should never exist in a program. However, in practice, there is some special reason for using write-only variables, as we shall see in Section 5.1. Single-reader-single-writer variables seem only having local interest, which usually are used inside a single module. Single-reader-multiple-writer and multiple-reader-single-writer variables are of greater interest, which usually are related to certain kind of system functionalities. Finally, multiple-reader-multiple-writer variables usually are related to functionalities that involve multiple modules of the system.

We modify Helgrind so that it can also record the access history of each shared variable. Appendix C describes some details on the modification we made to Helgrind. Figure 4 further shows the distributions of shared variables across different categories on workloads W1 and W2. The distributions are similar. Not surprisingly, most of the shared variables are multiple-reader-multiple-writer ones. However, there are also considerable number of variables that fall into other categories. In the rest of this section, we presents some typical examples in each category of roles of shared variables.

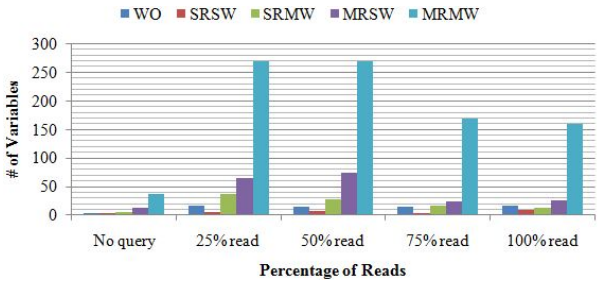
5.1 Write-Only

A typical example of write-only variables is the variable `rec_dummy` declared at line 145 of *rem0rec.c*[13].

[13] MySQL 5.5.11



(a) Distribution on W1.



(b) Distribution on W2.

Figure 4: Distribution of shared variables

The two statements that use this variable are: `rec_dummy = sum;`, appearing at line 1554 and 1617, respectively, where data races are reported.

The purpose for introducing this variable is to try to fool the compiler so that it will not do unexpected optimization to the code. Here, `sum` is a local variable that is never read. So a smart compiler will do an optimization by removing `sum` from the program, since it wastes CPU cycles for writing a *local* variable that is never read. However, the assignment statements (at line 1537 and 1600) to `sum` actually have other side effects that should not be bypassed. By assigning the value of `sum` to the global variable `rec_dummy`, the potential optimization by the compiler can be avoided.

5.2 Single-Reader-Single-Writer

A typical example of single-reader-single-writer variables is the variable `srv_lock_timeout_active` declared at line 107 of *srv0srv.c*. This variable is introduced to monitor whether there is some thread that waits too long on a lock. During a normal shutdown, the server should wait for other threads to stop first. Therefore `srv_lock_timeout_active` should be set to *false*. Since only the shutdown handler will read the value of `srv_lock_timeout_active`, while only the lock monitor will change the state of `srv_lock_timeout_active`, it is a single-reader-single-writer variable that only has local interest.

5.3 Single-Reader-Multiple-Writer

A typical example of single-reader-multiple-writer variables is the variable `shutdown_in_progress` declared at line 368 of *mysqld.cc*. This variable is introduced to indicate whether the server should be shut down. This variable is initialized to be *false*, and multiple threads can set it to be *true* if there is some severe event so that the server needs to be shut down. Only the shutdown handler will check the state of this variable to see whether the server should be shut down.

5.4 Multiple-Reader-Single-Writer

A typical example of multiple-reader-single-writer variables is the variable `srv_shutdown_state` declared at line 118 of *srv0start.c*. This variable can only be changed by the main thread during the shutdown procedure of the server. Meanwhile, the value of this variable will climb from `SRV_SHUTDOWN_NONE` to `SRV_SHUTDOWN_CLEANUP` and then to `SRV_SHUTDOWN_LAST_PHASE`, which indicate different stages in the shutdown process. Other threads will decide their own behavior on cleanup like releasing the locks that are held, by monitoring the value of `srv_shutdown_state`.

5.5 Multiple-Reader-Multiple-Writer

A typical example of multiple-reader-multiple-writer variables is the variable `os_n_file_reads` declared at line 281 of *os0file.c*. This variable counts the number of file reads performed so far by the server. Any thread that involves file reading will update this variable, and any thread needs this statistic should read this variable. In MySQL, threads that read this variable usually simply print it out for view.

6 Related Work

Automated data race detection tools typically deploy two different techniques: static or dynamic techniques. For static data race detection, some of the tools may look into the source code to locate potential races during compilation [1]. Some tools use type-based static analysis techniques, by adding new types to express synchronization operations, which can then be used by programs to prevent data races from happening [2]. Static analysis can also be done using model checking techniques [4].

For dynamic race detection, the memory accesses and the synchronization operations are recorded during the execution of a program. Then the records are analyzed to locate data races. Lockset [9] analysis and happens-before [5] analysis are used in dynamic race detectors.

Lockset based detectors verify that the execution of a program conforms to a locking discipline that ensures the absence of data races. Happens-before data race detectors are based on Lamport's happens-before relation [5]. The happens-before algorithm checks whether conflicting accesses to shared variables are ordered by synchronization operations or not. A data race occurs when two threads both access a shared variable and the accesses are not ordered by the happens-before relation. Many tools based on dynamic race detection have been developed, such as Eraser [8], LiteRace [6], and RaceTrack [10].

However, none of these algorithms are *perfect*. Lockset based algorithms usually report a much larger set of data races than those actually occur, resulting in many *false positives*. Happens-before based algorithms, on the other hand, will report less false positives. But they usually also cover less data race cases since they rely on predefined synchronization events, and hence can miss certain kinds of data races.

What's more, even if there is some perfect algorithm that can correctly find every data race, many of the data races detected are actually *benign* [7], i.e., they will not cause unexpected errors when running the program. Reporting a lot of benign races wastes programmers' labor on checking them and makes the task to locate real bugs more difficult. Motivated by this, the authors of [7] further propose a method that can automatically classify benign or harmful races. The idea is to replay the history but with different interleaving, and then check whether the memory content remains the same. However, there is no guarantee that the classification is correct.

7 Conclusion

In this paper, we studied the behavior of data races inside MySQL, a modern database management system. We first designed a benchmark consisting of SQL queries that are used in typical database applications, and collected a large set of data races by running the benchmark queries on MySQL. We then analyzed these races and identified a taxonomy of programming paradigms that can cause races. To learn the semantic implications of these paradigms, we further studied the roles of shared variables on which races are reported. Our findings reported in this paper will be helpful for programmers to get a deeper understanding on how to write more reliable multithreading programs.

As future work, we will investigate the aspect on roles of shared variables more deeply. Although our current study on their read/write histories reveals some interesting phenomena, more insights can be gained if we can interpret them at a higher level of semantics.

References

[1] ENGLER, D., AND ASHCRAFT, K. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev. 37* (October 2003), 237–252.

[2] FLANAGAN, C., AND FREUND, S. N. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 219–232.

[3] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. Race checking by context inference. *SIGPLAN Not. 39* (June 2004), 1–13.

[5] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[6] MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. Literace: effective sampling for lightweight data-race detection. *SIGPLAN Not. 44* (June 2009), 134–143.

[7] NARAYANASAMY, S., WANG, Z., TIGANI, J., EDWARDS, A., AND CALDER, B. Automatically classifying benign and harmful data racesallusing replay analysis. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 22–31.

[8] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15* (November 1997), 391–411.

[9] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. E. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP* (1997), pp. 27–37.

[10] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 221–234.

A The TPC-H Queries used in W3

Q1:

```
SELECT s_acctbal, s_name,
       n_name, p_partkey,
       p_mfgr, s_address,
       s_phone, s_comment
FROM part, supplier, partsupp,
     nation, region
WHERE p_partkey = ps_partkey
    AND s_suppkey = ps_suppkey
    AND p_size = 15
    AND p_type LIKE '%%brass'
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'europe'
    AND ps_supplycost = (
      SELECT MIN(ps_supplycost)
      FROM partsupp, supplier,
           nation, region
      WHERE p_partkey = ps_partkey
        AND s_suppkey = ps_suppkey
        AND s_nationkey = n_nationkey
        AND n_regionkey = r_regionkey
        AND r_name = 'europe')
ORDER BY s_acctbal DESC,
        n_name, s_name, p_partkey
```

Q2:

```
SELECT ps_partkey, SUM(ps_supplycost *
    ps_availqty) AS value
FROM partsupp, supplier, nation
WHERE ps_suppkey = s_suppkey
    AND s_nationkey = n_nationkey
    AND n_name = 'germany'
GROUP BY ps_partkey
HAVING SUM(ps_supplycost * ps_availqty)
        > (
     SELECT SUM(ps_supplycost *
            ps_availqty) * 0.0001
     FROM partsupp, supplier, nation
     WHERE ps_suppkey = s_suppkey
        AND s_nationkey = n_nationkey
        AND n_name = 'germany' )
ORDER BY value DESC
```

Q3:

```
SELECT p_brand, p_type, p_size,
    COUNT(DISTINCT ps_suppkey) AS
        supplier_cnt
FROM partsupp, part
WHERE p_partkey = ps_partkey
    AND p_brand <> 'brand#45'
    AND p_type NOT LIKE 'medium
                  polished%%'
    AND p_size IN (49, 14, 23, 45,
             19, 3, 36, 9)
    AND ps_suppkey NOT IN (
        SELECT s_suppkey
        FROM supplier
        WHERE s_comment LIKE '%%customer
                %%complaints%%' )
GROUP BY p_brand, p_type, p_size
ORDER BY supplier_cnt DESC, p_brand,
        p_type, p_size
```

B User Defined Mutex in MySQL

In this section, we study the implementation details of the user defined synchronization primitives `mutex_enter()` and `mutex_exit()` in MySQL.

The code snippet of `mutex_enter()` is as follows:

```
mutex_enter(){
   if (!mutex_test_and_set(mutex)) {
     mutex->thread_id
            = os_get_thread_id();
   }
   mutex_spin_wait(mutex,
            file_name, line);
}
```

, where `mutex_test_and_set()` is defined as:

```
mutex_test_and_set(mutex_t* mutex)
{
#if defined(HAVE_ATOMIC_BUILTINS)
    atomic_test_and_set_byte(
            &mutex->lock_word,1);
#else
    bool ret;
    ret = os_fast_mutex_trylock(
        &(mutex->os_fast_mutex));
    if (ret == 0) {
        ut_a(mutex->lock_word == 0);
        mutex->lock_word = 1;
    }
    return(ret);
#endif
}
```

In `mutex_enter()`, `mutex_test_and_set()` grabs the mutex if it is available. Otherwise, the thread has to `spin_wait`. In `mutex_test_and_set()`, there are two compilation options, based on different platforms. The first option is an atomic instruction `test_and_set`, which should be supported by hardware. The second option is to use the atomic function `mutex_trylock()` to check whether the lock

is available or not. If it is available, `ret` is set to be 0. However, before the thread grabs the lock, the `ut_a()` assertion is used to double check that the mutex is not stolen by another thread. Because it is possible that after the statement `if (ret == 0)` is executed, the current thread is preempted and the preempting thread gets the lock and sets `mutex->lock_word` to be 1. If this is the case, the assertion in `ut_a()` fails and the current thread will be aborted. Though data races do exist on `mutex->lock_word`, the abortion mechanism ensures that only one thread can get the mutex, and hence there are no data races on the variable `mutex->thread_id`.

Interestingly, Helgrind will report races on `mutex->thread_id`, since again it does not know that `mutex_test_and_set()` is also a user constructed synchronization primitive.

The function `mutex_exit()` is defined as:

```
mutex_exit(){
   ut_ad(mutex_own(mutex));
   mutex->thread_id = ULINT_UNDEFINED;
   mutex_reset_lock_word(mutex);
}
```

In `mutex_exit()`, the `thread_id` of the mutex is reset to `ULINT_UNDEFINED` after the thread exits. However, data race happens when another thread preempts, calls `mutex_enter()` and updates the `mutex->thread_id`. This race has no effect on the correctness, since the `thread_id` will finally be assigned by the new thread even if the preemption does not occur.

C Modification to Helgrind

We modified Helgrind so that it can also record the access histories of shared variables. In this section we discuss the details of our implementation.

Helgrind can intercept each `load` and `store` instruction in the execution of a binary program. Helgrind then registers it's own *hook* functions `msmcread` / `msmcwrite` (defined in *libhb_core.c*) that will be called whenever a `load` / `store` instruction is executed. Originally Helgrind will check and record possible data races inside these two functions.

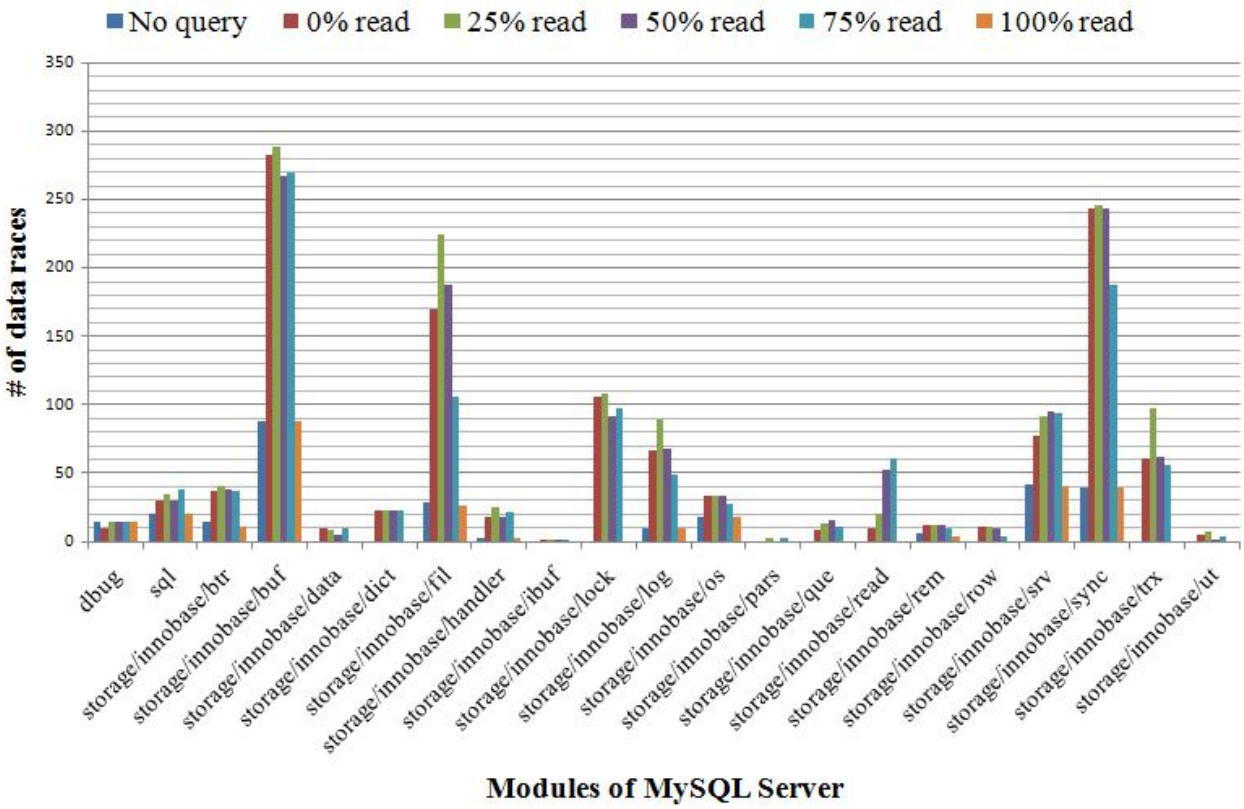
To record the information for shared variables, we add our own functions and call them before returning from `msmcread` / `msmcwrite`. The first problem we face is how we can know whether an address is accessed by more than one thread. Unfortunately, there seems no better way than the simplest idea to keep track of every address that is accessed. For the purpose of quick lookup, we use a hashtable to store the addresses. For each address, we keep a list of threads that have accessed it. If the list of an address contains more than one thread, then we know that this address is shared.

The second problem is that what information we should record for each thread. For our purpose on classifying shared variables based on their access histories, we only record the number of reads and writes made by each thread.

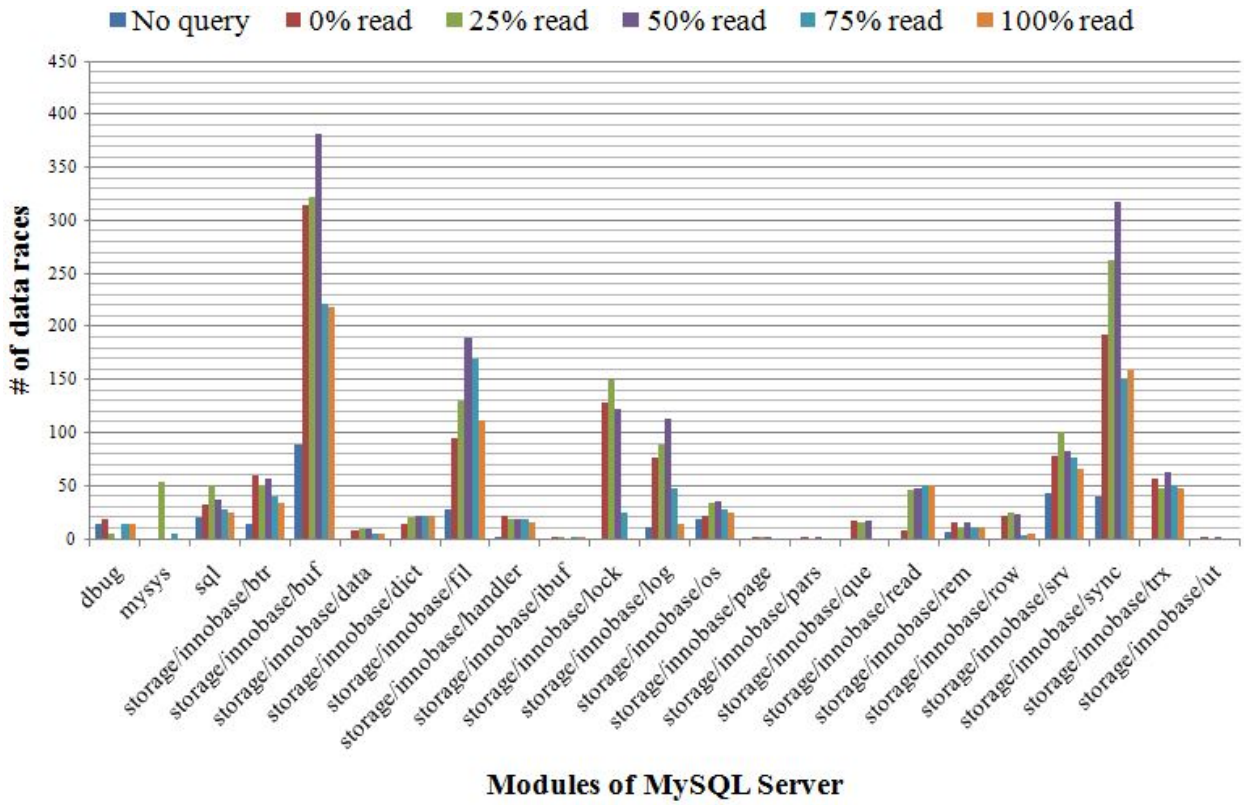
The last problem is that when we should output the information recorded. If we record every read/write event instead of accumulated statistics, we can output them immediately after they are intercepted. However, this will generate a huge log file on the disk, and the number of disk I/O's involved is also considerable. This is the main motivation for us to only record aggregated information. But then we have to choose appropriate moments to output the statistics. We decide to dump out the information recorded so far for a shared address when a *new thread* reads/writes it. In this way, we can preserve the complete information we need to categorize variables based on the access histories.

D Locations of Functions

1. `fil_flush_file_spaces`: *fil0fil.c*, line 4701.
2. `fil_flush`: *fil0fil.c*, line 4663.
3. `srv_release_threads`: *srv0srv.c*, line 940.
4. `srv_thread_has_reserved_slot`: *srv0srv.c*, line 1013.
5. `buf_pool_check_no_pending_io`: *buf0buf.c*, line 5196.
6. `buf_page_io_complete`: *buf0buf.c*, line 4092.
7. `logs_empty_and_mark_files_at_shutdown`: *log0log.c*, line 3089.
8. `srv_lock_timeout_thread`: *srv0srv.c*, line 2340.
9. `btr_cur_search_to_nth_level`: *btr0cur.c*, line 500.
10. `srv_refresh_innodb_monitor_stats`: *srv0srv.c*, line 1798.
11. `os_file_read_func`: *os0file.c*, line 2487.
12. `os_aio_refresh_stats`: *os0file.c*, line 5181.
13. `trx_assign_rseg`: *trx0trx.c*, line 615.
14. `mutex_enter`: *sync0sync.ic*, line 200.
15. `mutex_test_and_set`: *sync0sync.ic*, line 78.
16. `mutex_exit`: *sync0sync.ic*, line 159.

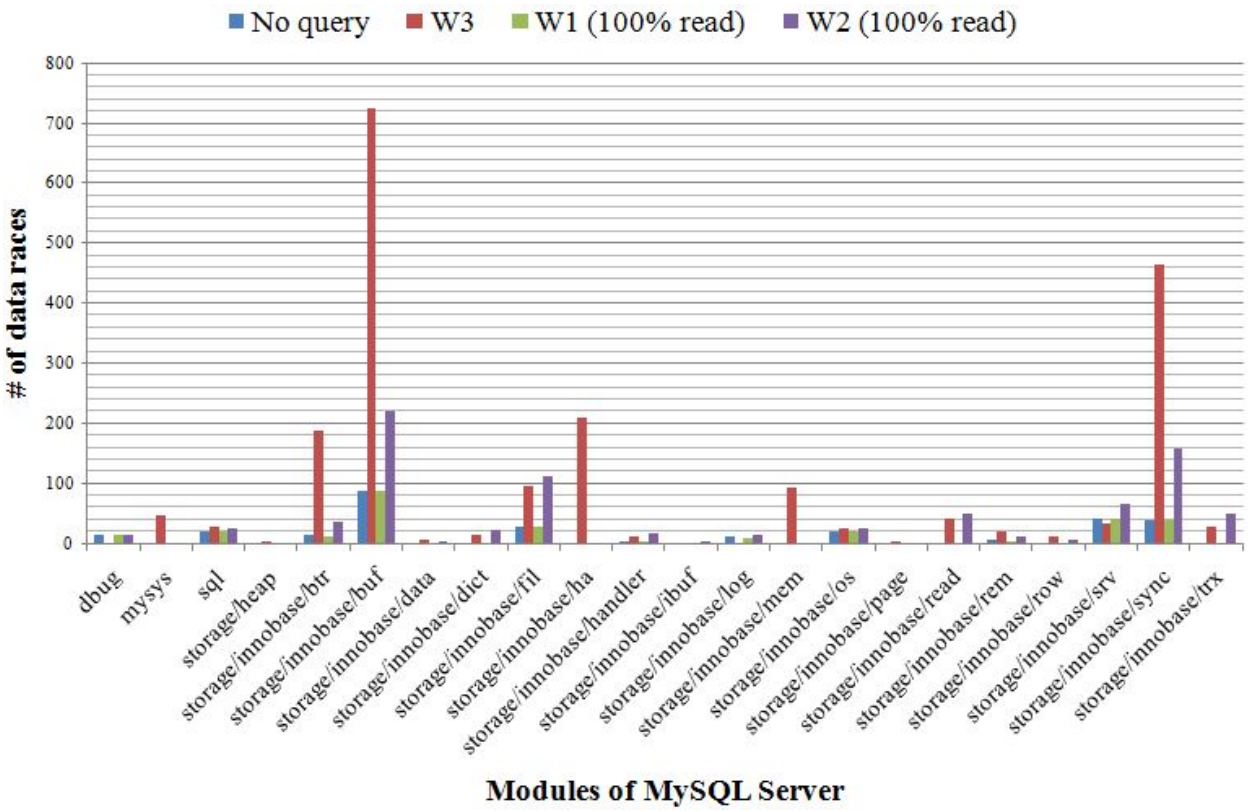


(a) Distribution on W1.

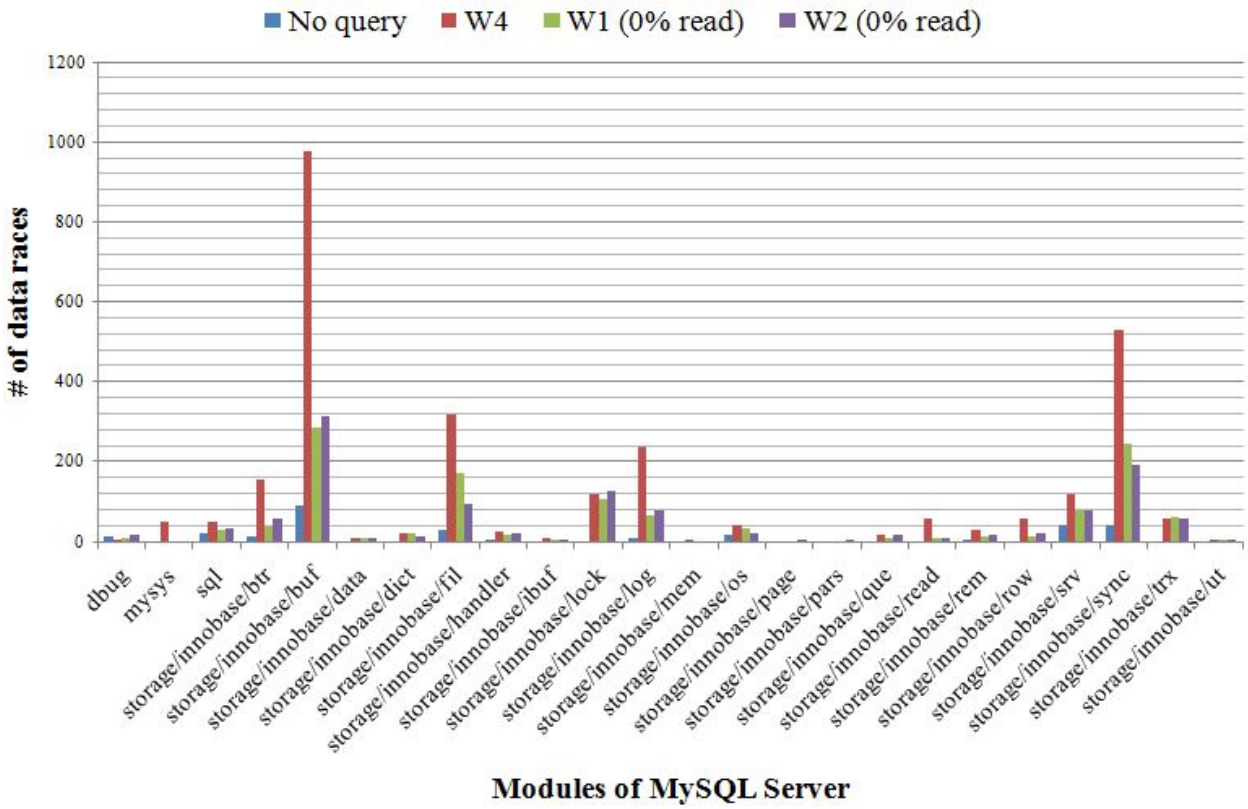


(b) Distribution on W2.

Figure 2: Distribution of data races across different modules in MySQL on W1 and W2



(a) Distribution on W3.



(b) Distribution on W4.

Figure 3: Distribution of data races across different modules in MySQL on W3 and W4