Chronicle Map Research Project

Dabeer Ahmed

October 2024

1 Chronicle Map

Chronicle Map seemed an interesting project for my analysis of concurrency due to its relevance to high-performance, low-latency systems such as *High-Frequency Trading (HFT)*, a field I am deeply interested in pursuing. HFT systems demand the ability to process large volumes of data in real-time, and any delays in managing concurrent operations can lead to missed trading opportunities. *Chronicle Map* is specifically designed for environments that require efficient handling of concurrent read and write operations while maintaining ultra-low latency. Its key features, such as off-heap memory storage and synchronized access to shared resources, are aligned with the real-time requirements of HFT, making it an ideal project for my concurrency analysis using the *Facebook Infer* tool.

2 Benefits of Concurrency in Chronicle Map

There are 2 files that utilize concurrency heavily, named *ChronicleHashResources* and *CompiledMapQueryContext*. For brevity, I will focus on *ChronicleHashResources* for this project. From my understanding, *ChronicleHashResources* employs concurrency to ensure efficient access to shared resources like `contexts`, `memoryResources`, and `closeables` by multiple threads. By utilizing synchronized blocks in critical sections where writes occur, *Chronicle Map* prevents race conditions while enabling non-blocking reads, thereby ensuring that performance remains optimal under heavy loads. The system supports multiple processes accessing shared memory without blocking, which is particularly beneficial for distributed and multi-threaded applications. Additionally, its use of off-heap memory minimizes garbage collection overhead, making it particularly suitable for performance-critical applications such as HFT, where speed and consistency are vital.

Moreover, concurrency plays a key role in the *resource cleanup process* through the use of the `Cleaner`, which allows resources to be released asynchronously without interrupting ongoing operations. This asynchronous cleanup process

is crucial for systems that require continuous, real-time performance, as it allows resource management to happen in the background without blocking other tasks.

3 Results of Facebook Infer

Running Infer on the project yielded a total of 19 issues, and I focused on the ones that were yielded from *ChronicleHashResources*, which were 8.

4 8 Detected Thread Safety Violations

Issue #0: Thread Safety Violation in `contexts()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:61: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.contexts()‘ reads without
synchronization from ‘this.contexts‘, which races with the write in method
‘ChronicleHashResources.run()‘. Reporting because another access to the same memory occurs
on a background thread, although this access may not.
59.
60.       List<WeakReference<ContextHolder>> contexts() {
61. >         return contexts;
62.       }
63.
```

Issue #1: Thread Safety Violation in `totalMemory()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:69: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.totalMemory()‘ reads with
synchronization from ‘this.memoryResources‘, which races with the unsynchronized write
in method ‘ChronicleHashResources.releaseManually()‘. Reporting because this access may
occur on a background thread.
67.           long totalMemory = 0L;
68.           //noinspection ForLoopReplaceableByForEach -- allocation-free looping
69. >         for (int i = 0; i < memoryResources.size(); i++) {
70.               totalMemory += memoryResources.get(i).size;
71.           }
```

Issue #2: Thread Safety Violation in `addMemoryResource()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:107: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.addMemoryResource(...)‘
reads with synchronization from ‘this.memoryResources‘, which races with the unsynchronized
```

```
write in method ‘ChronicleHashResources.releaseManually()‘. Reporting because this access
may occur on a background thread.
105.           synchronized (this) {
106.               checkOpen();
107. >             memoryResources.add(new MemoryResource(address, size));
108.           }
109.       }
```

Issue #3: Thread Safety Violation in `addCloseable()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:116: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.addCloseable(...)‘ reads
with synchronization from ‘this.closeables‘, which races with the unsynchronized write
in method ‘ChronicleHashResources.releaseManually()‘. Reporting because this access
may occur on a background thread.
114.           synchronized (this) {
115.               checkOpen();
116. >             closeables.add(closeable);
117.           }
118.       }
```

Issue #4: Thread Safety Violation in `addContext()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:125: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.addContext(...)‘ indirectly
reads with synchronization from ‘this.contexts‘, which races with the unsynchronized
write in method ‘ChronicleHashResources.releaseManually()‘. Reporting because this access
may occur on a background thread.
123.           synchronized (this) {
124.               checkOpen();
125. >             expungeStateContexts();
126.               contexts.add(new WeakReference<>(contextHolder));
127.           }
```

Issue #5: Thread Safety Violation in `run()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:156: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.run()‘ indirectly reads
with synchronization from ‘this.closeables‘, which races with the unsynchronized
write in method ‘ChronicleHashResources.releaseManually()‘. Reporting because this access
may occur on a background thread.
154.                                       "this should be impossible");
155.                       } else {
```

```
156. >                          thrown = Throwables.returnOrSuppress(thrown,
                                 releaseEverything(true));
157.                        }
158.                    }
```

Issue #6: Thread Safety Violation in `run()` with `chronicleHashIdentityString`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:161: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.run()‘ reads without
synchronization from ‘this.chronicleHashIdentityString‘, which races with the write in
method ‘ChronicleHashResources.setChronicleHashIdentityString(...)‘. Reporting because
this access may occur on a background thread.
159.                    if (thrown != null) {
160.                        try {
161. >                          Jvm.error().on(getClass(),
                                 "Error on releasing resources of " + chronicleHashIdentitySt
162.                                    thrown);
163.                        } catch (Throwable t) {
```

Issue #7: Thread Safety Violation in `releaseManually()`

```
src/main/java/net/openhft/chronicle/hash/impl/ChronicleHashResources.java:202: warning:
Thread Safety Violation(THREAD_SAFETY_VIOLATION)
Read/Write race. Non-private method ‘ChronicleHashResources.releaseManually()‘
indirectly reads without synchronization from ‘this.chronicleHashIdentityString‘, which
races with the write in method ‘ChronicleHashResources.setChronicleHashIdentityString(...)‘.
Reporting because this access may occur on a background thread.
200.            }
201.
202. >          Throwable thrown = releaseEverything(false);
203.            if (thrown != null)
204.                throw Throwables.propagate(thrown);
```

Why Infer Detects Issue with Issue #0 and #4

Infer identifies a possible data race in the method `addContext(ContextHolder contextHolder)`, where a new `ContextHolder` is added to the `contexts` list. This method operates within a synchronized block to ensure that modifications to the list, including the addition of new elements, are thread-safe within this specific method. However, there is another method, `releaseManually()`, which accesses the `contexts` list without synchronization. This unsynchronized access introduces a risk of a race condition, where one thread might be modifying the list through `addContext()` while another thread modifies the same list concurrently in `releaseManually()`. Such simultaneous operations can lead to data

inconsistencies, prompting Infer to flag this as a potential issue.

How Declaring `contexts` as Volatile Fixes This

Declaring the `contexts` list as **volatile** addresses this race condition by ensuring that any modifications made to `contexts` by one thread are immediately visible to all other threads. The volatile declaration guarantees that even when `releaseManually()` and `addContext()` are executed concurrently, the most up-to-date state of the `contexts` list is accessed by all threads. This eliminates the risk of data inconsistencies without requiring additional synchronization, allowing safe concurrent access to the list while maintaining performance. By making `contexts` volatile, we ensure proper visibility and ordering of operations, preventing race conditions and ensuring thread safety.

Why Infer Detects Issue #1 and #2

Infer identifiesan issue which occurs during the addition of an element to `memoryResources` via the `add()` method. This issue arises because `memoryResources.add()` involves both a **read** operation to check the current state of the list and a **write** operation to append the new element. The tool detects this race condition since, while one thread reads from the list to perform the add operation, another thread could concurrently be modifying the same list through the `releaseManually()` method. This method, by calling `releaseEverything()`, invokes `releaseMemoryResources()`, which writes to the `memoryResources` list. The simultaneous read and write operations from different threads can result in inconsistencies, prompting Infer to flag the issue.

How Declaring `memoryResources` as Volatile Fixes This

To address this potential conflict, `memoryResources` is declared as **volatile**. Declaring the variable as volatile ensures that any changes made by one thread are immediately visible to all other threads, preventing threads from reading outdated or partially updated data. By making `memoryResources` volatile, we enforce a **sequential consistency** between read and write operations, ensuring that all threads access the most up-to-date version of the list. This avoids the race condition by ensuring that threads cannot perform conflicting operations simultaneously, maintaining both safety and performance without requiring explicit synchronization.

Interestingly, making both contexts and memoryResources volatile fixes Issues 10, 13, and 17 from the other file *CompiledMapQueryContext*. However, these are reintroduced after fixing **Issue #3** and **Issue #5**

Why Infer Detects Issue with Issue #3 and #5

Infer detects a potential data race in the method `addCloseable(Closeable closeable)`, which adds a `Closeable` object to the `closeables` list. The issue arises because `closeables.add()` is a write operation that occurs within a synchronized block, ensuring that only one thread can modify the list at a time in this method. However, there is another method, `releaseManually()`, which can modify the `closeables` list without synchronization. If `releaseManually()` writes to `closeables` concurrently while `addCloseable()` is reading or modifying the same list, a race condition occurs. This is why Infer flags the issue, as the simultaneous access to the list by different threads can lead to inconsistent states.

How Declaring `closeables` as Volatile Fixes This

To prevent this race condition, the `closeables` list is declared as **volatile**. Making `closeables` volatile ensures that any updates made to it by one thread are immediately visible to other threads. This guarantees that even when `releaseManually()` modifies the list concurrently with `addCloseable()`, all threads have access to the most up-to-date version of the list. By enforcing **visibility** of changes across threads, the volatile declaration eliminates the race condition, ensuring that concurrent operations on `closeables` are performed safely without the need for additional synchronization.

Why Infer Detects Issue #7

Issue #7 arises due to a race condition involving the shared variable `chronicleHashIdentityString`. In the method `ChronicleHashResources.releaseManually()`, this variable is read without synchronization, while in the method `setChronicleHashIdentityString(...)`, the same variable is written. Infer detects this as a potential read/write race condition because both read and write operations can happen concurrently in a multi-threaded environment, leading to inconsistent or stale data.

How Making `releaseManually()` Synchronized Fixes This

The race condition is fixed by making the `releaseManually()` method **synchronized**. Declaring the method as synchronized ensures that only one thread can access `releaseManually()` at a time, preventing concurrent modification of `chronicleHashIdentityString` by other threads. This guarantees safe access and avoids potential inconsistencies caused by simultaneous read and write operations on the variable. The method is now declared as:

```
public synchronized final boolean releaseManually()
```

This ensures that all accesses to `chronicleHashIdentityString` within `releaseManually()` are thread-safe.

Why Infer Detects Issue #6

Finally, **Issue #6** is detected due to a race condition involving the variable `chronicleHashIdentityString`. In the method `ChronicleHashResources.run()`, this variable is read without synchronization, while in the method `setChronicleHashIdentityString(...)`, the same variable is written to. Infer detects this as a potential **read/write race condition** because both read and write operations can happen concurrently on separate threads, leading to inconsistent or stale data during execution.

How Declaring `chronicleHashIdentityString` as Volatile Fixes This

The race condition is resolved by declaring the variable `chronicleHashIdentityString` as **volatile**. By making the variable volatile, all threads will see the latest value written to it immediately, ensuring that no stale or inconsistent data is accessed during read operations. The variable is now declared as:

```
private volatile String chronicleHashIdentityString;
```

This guarantees that the read in `run()` and the write in `setChronicleHashIdentityString(...)` are safely visible across all threads, eliminating the race condition.

5 More Exploration

Interestingly, synchronizing `releaseManually()` and making `contexts` and `chronicleHashIdentityString` volatile were enough to solve all the problems in the `ChronicleHashResources`, and also resolved a few issues (3) in `CompiledMapQueryContext`, reducing the leftover issues to 8. However, the issues in `CompiledMapQueryContext` were reintroduced after `MemoryResource` and `Closeable` were also made volatile. This could signify a more subtle concurrency problem where thread coordination and operation reordering may need more explicit synchronization. Alternatively, it might indicate a limitation or flaw in the Infer tool itself. While Infer is effective at identifying potential race conditions, it may sometimes misinterpret or over-report issues when volatile variables are involved, especially in more complex, highly concurrent systems. This raises the possibility that the tool's analysis may not fully capture the context of how threads interact in these scenarios, leading to the reintroduction of errors that might not be problematic in practice. A closer look at Infer's analysis logic or alternative concurrency testing tools could help clarify whether this is a true race condition or a false positive from the tool. However, the scope of this research is limited and leaves room for further exploration.